

MPLAB® C30
C 编译器
用户指南

请注意以下有关 Microchip 器件代码保护功能的要点：

- Microchip 的产品均达到 Microchip 数据手册中所述的技术指标。
- Microchip 确信：在正常使用的情况下， Microchip 系列产品是当今市场上同类产品中最安全的产品之一。
- 目前，仍存在着恶意、甚至是非法破坏代码保护功能的行为。就我们所知，所有这些行为都不是以 Microchip 数据手册中规定的操作规范来使用 Microchip 产品的。这样做的人极可能侵犯了知识产权。
- Microchip 愿与那些注重代码完整性的客户合作。
- Microchip 或任何其他半导体厂商均无法保证其代码的安全性。代码保护并不意味着我们保证产品是 “牢不可破”的。

代码保护功能处于持续发展中。 Microchip 承诺将不断改进产品的代码保护功能。任何试图破坏 Microchip 代码保护功能的行为均可视为违反了 《数字器件千年版权法案 （Digital Millennium Copyright Act）》。如果这种行为导致他人在未经授权的情况下，能访问您的软件或其他受版权保护的成果，您有权依据该法案提起诉讼，从而制止这种行为。

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
== ISO/TS 16949:2002 ==

Microchip 位于美国亚利桑那州 Chandler 和 Tempe、位于俄勒冈州 Gresham 及位于加利福尼亚州 Mountain View 的全球总部、设计中心和晶圆生产厂均通过了 ISO/TS-16949:2002 认证。公司在 PICmicro® 8 位单片机、 KEELOQ® 跳码器件、串行 EEPROM、单片机外设、非易失性存储器和模拟产品方面的质量体系流程均符合 ISO/TS-16949:2002。此外， Microchip 在开发系统的设计和生产方面的质量体系也已通过了 ISO 9001:2000 认证。



目录



前言

客户须知

所有文档均会过时，本文档也不例外。 **Microchip** 的工具和文档将不断演变以满足客户的需求，因此实际使用中有些对话框和 **/** 或工具说明可能与本文档所述之内容有所不同。请访问我们的网站（**www.microchip.com**）获取最新文档。

文档均标记有 “**DS**”编号。该编号出现在每页底部的页码之前。 **DS** 编号的命名约定为“**DSXXXXXA**”，其中 “**XXXXX**”为文档编号，“**A**”为文档版本。

欲了解开发工具的最新信息，请参考 **MPLAB® IDE** 在线帮助。从 **Help** （帮助）菜单选择 **Topics** （主题），打开现有在线帮助文件列表。

简介

本文档的目的是帮助大家使用 Microchip 针对 dsPIC® 数字信号控制器（Digital Signal Controllers， DSC）的 MPLAB C30 C 编译器开发应用程序。MPLAB C30 是一款基于 GNU 编译器集 （GNU Compiler Collection， GCC）的语言工具，以来自 Free Software Foundation （FSF）的源代码为基础。关于 FSF 的信息可登陆网站 WWW.FSF.ORG。

您还可从 Microchip 获得其他 GNU 语言工具：

- MPLAB ASM30 汇编器
- MPLAB LINK30 链接器
- MPLAB LIB30 库管理器 / 归档程序

本章涉及以下内容：

- 关于本指南
- 推荐读物
- 疑难解答
- Microchip 网站
- 开发系统变更通知客户服务
- 客户支持

关于本指南

文档编排

本文档介绍如何使用 MPLAB C30 来开发固件。内容安排如下：

- 第 **1** 章：**编译器概述** — 介绍 MPLAB C30、开发工具和功能。
- 第 **2** 章：**MPLAB C30 与 ANSI C 的区别** — 描述 MPLAB C30 语法支持的 C 语言和标准 ANSI-89 C 之间的区别。
- 第 **3** 章：**使用 MPLAB C30** — 介绍怎样通过命令行使用 MPLAB C30。
- 第 **4** 章：**MPLAB C30 运行时环境** — 介绍 MPLAB C30 运行时模型，包括段信息、初始化、存储器模式和软件堆栈等。
- 第 **5** 章：**数据类型** — 介绍 MPLAB C30 整型、浮点型和指针型数据类型。
- 第 **6** 章：**器件支持文件** — 介绍 MPLAB C30 的头文件和寄存器定义文件，以及怎样使用特殊功能寄存器。
- 第 **7** 章：**中断** — 介绍怎样使用中断。
- 第 **8** 章：**C 语言与汇编语言的混合编程** — 为使用 MPLAB C30 与 MPLAB ASM30 汇编语言模块提供指导。
- 附录 **A**：**实现定义的操作** — 详细描述 ANSI 标准中描述为实现定义的、特定于 MPLAB C30 的参数。
- 附录 **B**：**MPLAB C30 诊断信息** — 列出由 MPLAB C30 产生的错误和警告消息。
- 附录 **C**：**MPLAB C18 和 MPLAB C30 的区别** — 介绍 PIC18XXXXX 编译器（MPLAB C18）和 dsPIC DSC 编译器 （MPLAB C30）的主要区别。
- 附录 **D**：**不赞成使用的特性** — **详述视为过时的特性。**
- 附录 **E**：**ASCII 字符集** — 介绍 ASCII 字符集。
- 附录 **F**：**GNU 自由文档许可证** — Free Software Foundation 的使用许可证。

本文档中使用的约定

本手册使用如下文档约定：

文档约定

说明	涵义	示例
Arial 字体：		
斜体字	参考书目	*MPLAB® IDE User's Guide*
	需强调的文字	... *仅有*的编译器 ...
首字母大写	窗口	Output 窗口
	对话框	Settings 对话框
	菜单选项	选择 Enable Programmer
引号	窗口或对话框中的字段名	“Save project before build”
带右尖括号且有下划线的斜体文字	菜单路径	*File>Save*
粗体字	对话框按钮	单击 **OK**
	选项卡	单击 **Power** 选项卡
‘b*nnnn*	二进制数， *n* 是一个数字	‘b00100, ‘b10
尖括号 < > 括起的文字	键盘上的按键	按 <Enter>，<F1>
Courier 字体：		
常规 Courier New	源代码示例	`#define START`
	文件名	`autoexec.bat`
	文件路径	`c:\mcc18\h`
	关键字	`_asm, _endasm, static`
	命令行选项	`-Opa+, -Opa-`
	位值	`0, 1`
	常数	`0xFF, 'A'`
斜体 Courier New	可变参数	*file*.o，其中 *file* 可以是任一有效文件名
0x*nnnn*	十六进制数， *n* 是一个十六进制数字	0xFFFF，0x007A
方括号 [ ]	可选参数	`mcc18 [options]` *file* `[options]`
花括号和竖线：{ | }	选择互斥参数；“或”选择	`errorlevel {0|1}`
省略号 ...	代替重复文字	`var_name [, var_name...]`
图标		
Full Version Only	仅完全版软件支持的功能。	
☆	并非所有的器件都支持此功能。所支持器件将在标题或正文中列出。	

推荐读物

本文档介绍如何使用 MPLAB C30 编译器。下面列出了其他有用的文档。 Microchip 提供了如下文档，推荐将这些文档作为补充参考资料。

README 文件

关于 Microchip 工具的最新信息，请阅读软件附带的相关 README 文件 （ASCII 文本文件）。

dsPIC® DSC 语言工具入门 （DS70094C_CN）

指导安装和使用 Microchip dsPIC 数字信号控制器 （DSC）的语言工具 （MPLAB ASM30、 MPLAB LINK30 和 MPLAB C30）。还提供了使用 dsPIC DSC 软件模拟器 MPLAB SIM30 的示例。

MPLAB® ASM30、 MPLAB LINK30 和实用程序用户指南 （DS51317D_CN）

指导使用 dsPIC DSC 汇编器、MPLAB ASM30、dsPIC DSC 链接器、MPLAB LINK30 和各种 dsPIC DSC 实用程序，包括 MPLAB LIB30 归档程序 / 库管理器。

dsPIC30F Data Sheet General Purpose and Sensor Families （DS70083）

这是 dsPIC30F 数字信号控制器 （DSC）的数据手册。总体介绍了器件及其架构。详细介绍了存储器的构成、 DSP 操作和外设功能。还包括器件的电气参数。

dsPIC30F 系列参考手册 （DS70046D_CN）

该系列的参考指南，介绍了 dsPIC30F 系列 DSC 的架构和外设。

dsPIC30F/33F 程序员参考手册 （DS70157B_CN）

dsPIC30F/33F 器件的编程人员指南。包括编程模型和指令集。

C 标准信息

American National Standard for Information Systems – *Programming Language – C*. American National Standards Institute (ANSI), 11 West 42nd. Street, New York, New York, 10036.

此标准规定了用 C 语言编写程序的格式，并对 C 程序进行了解释。其目的是提高 C 程序在多种计算机系统上的可移植性、可靠性、可维护性及执行效率。

C 参考书籍

Harbison, Samuel P., and Steele, Guy L., *C A Reference Manual*，第四版，Prentice-Hall, Englewood Cliffs, N.J. 07632.

Kernighan, Brian W., and Ritchie, Dennis M., *The C Programming Language*，第二版。 Prentice Hall, Englewood Cliffs, N.J. 07632.

Kochan, Steven G., *Programming In ANSI C*，修订版。 Hayden Books, Indianapolis, Indiana 46268.

Plauger, P.J., *The Standard C Library*, Prentice-Hall, Englewood Cliffs, N.J. 07632.

Van Sickle, Ted., *Programming Microcontrollers in C*，第一版。 LLH Technology Publishing, Eagle Rock, Virginia 24085.

疑难解答

在本文中找不到的常见问题信息可以查阅 README 文件。

MICROCHIP 网站

Microchip 网站 （www.microchip.com）为客户提供在线支持。客户可通过该网站方便地获取文件和信息。只要使用常用的因特网浏览器即可访问。网站提供以下信息：

- **产品支持**——数据手册和勘误表、应用笔记和样本程序、设计资源、用户指南以及硬件支持文档、最新的软件版本以及存档软件
- **一般技术支持**——常见问题 （FAQ）、技术支持请求、在线讨论组以及 Microchip 顾问计划成员名单
- **Microchip 业务**——产品选型和订购指南、最新 Microchip 新闻稿、研讨会和活动安排表、 Microchip 销售办事处、代理商以及工厂代表列表

开发系统变更通知客户服务

Microchip 的客户通知服务有助于客户了解 Microchip 产品的最新信息。注册客户可在他们感兴趣的某个产品系列或开发工具发生变更、更新、发布新版本或勘误表时，收到电子邮件通知。

欲注册，请登录 Microchip 网站 www.microchip.com，点击 “变更通知客户（Customer Change Notification）” 服务并按照注册说明完成注册。

开发系统产品的分类如下：

- **编译器**——Microchip C 编译器及其他语言工具的最新信息，包括 MPLAB C18 和 MPLAB C30 C 编译器、 MPASM™ 和 MPLAB ASM30 汇编器、 MPLINK™ 和 MPLAB LINK30 目标链接器，以及 MPLIB™ 和 MPLAB LIB30 目标库管理器。
- **仿真器**——Microchip 在线仿真器的最新信息，包括 MPLAB ICE 2000 和 MPLAB ICE 4000。
- **在线调试器**——Microchip 在线调试器 MPLAB ICD 2 的最新信息。
- **MPLAB® IDE**——关于支持开发系统工具的 Windows® 集成开发环境 Microchip MPLAB IDE 的最新信息，主要针对 MPLAB IDE、 MPLAB SIM 模拟器、 MPLAB IDE 项目管理器以及一般编辑和调试功能。
- **编程器**——Microchip 编程器的最新信息，包括 MPLAB PM3 和 PRO MATE® II 器件编程器以及 PICSTART® Plus 和 PICkit® 1 开发编程器。

客户支持

Microchip 产品的用户可通过以下渠道获得帮助：

- 代理商或代表
- 当地销售办事处
- 应用工程师 （FAE）
- 技术支持

客户应联系其代理商、代表或应用工程师 （FAE）寻求支持。当地销售办事处也可为客户提供帮助。本文档后附有销售办事处的联系方式。

也可通过 http://support.microchip.com 获得网上技术支持。



第 1 章　编译器概述

1.1　简介

dsPIC® 数字信号控制器 （DSC）系列将 DSP 应用所需的高性能和嵌入式应用所需的标准单片机功能融合在一起。也可以将不带 DSP 功能的其他高性能单片机 （MCU）用于其他应用。

所有这些器件得到了一套完整的软件开发工具的充分支持，包括一个优化的 C 编译器、一个汇编器、一个链接器和一个归档程序 / 库管理器。

本章总体介绍了这些工具以及优化的 C 编译器的功能，包括 C 编译器如何与 MPLAB ASM30 汇编器和 MPLAB LINK30 链接器配合工作。汇编器和链接器在 *《MPLAB® ASM30*、 *MPLAB LINK30 和实用程序用户指南》*（DS51317D_CN）中有详细介绍。

1.2　主要内容

本章将介绍如下内容：

- MPLAB C30 介绍
- MPLAB C30 及其他开发工具
- MPLAB C30 的功能

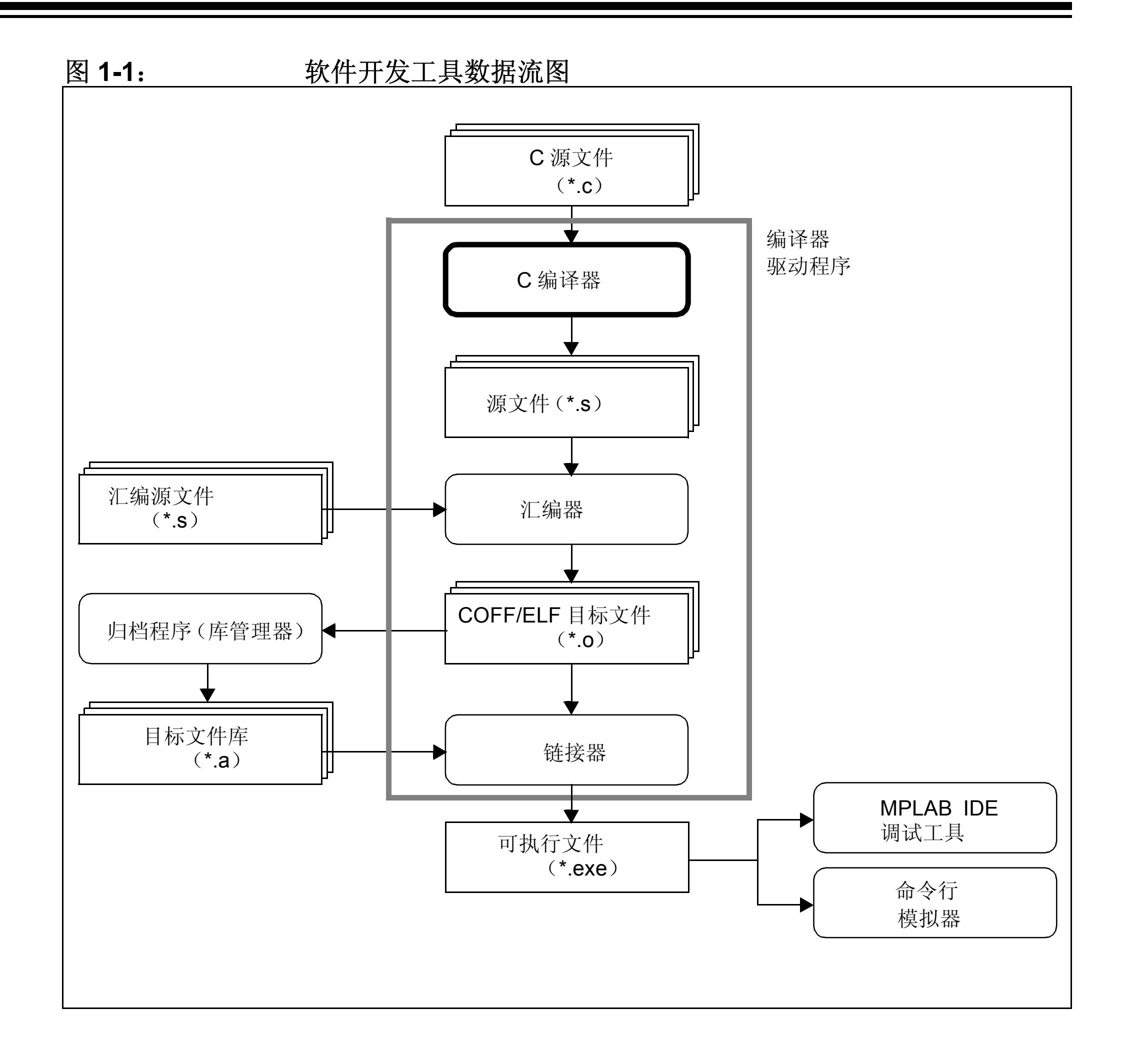
1.3　MPLAB C30 介绍

MPLAB C30 是一个遵循 ANSI x3.159-1989 标准的优化 C 编译器，它包括针对 dsPIC DSC 嵌入式控制应用的语言扩展。这个编译器是基于 Windows® 操作系统的应用程序，它为开发 C 代码提供了一个平台。它是来自 Free Software Foundation 的 GCC 编译器。

1.4　MPLAB C30 及其他开发工具

MPLAB C30 对 C 源文件进行编译，生成汇编语言文件。由编译器生成的文件与其他目标文件和库文件进行汇编和链接以产生最终的应用程序，应用程序格式为 COFF 或 ELF 文件格式。 COFF 或 ELF 文件可以载入 MPLAB IDE 中进行测试和调试，或使用转换实用程序将 COFF 或 ELF 文件转换为 Intel® hex 格式，以便载入命令行模拟器或器件编程器。图 1-1 概括了软件开发的数据流图。

图 **1-1：** 软件开发工具数据流图



1.5 MPLAB C30 的功能

MPLAB C30 C 编译器是一个全功能的优化编译器，可将标准的 ANSI C 程序翻译为 dsPIC DSC 汇编语言源代码。同时它还支持许多命令行选项和语言扩展，可以对 dsPIC DSC 器件的硬件功能进行完全访问，以便更好地控制代码的生成。这一节介绍了编译器的主要功能。

1.5.1 ANSI C 标准

MPLAB C30 编译器是一个完全经过验证的编译器，符合 ANSI C 标准，该标准由 ANSI 规范定义并在 Kernighan 和 Ritchie 的 *C Programming Language* （第二版）中有详细说明。 ANSI 标准包括对原始 C 定义的扩展，这些扩展现在成为 C 语言的标准功能。这些扩展增强了 C 语言的可移植性，同时增强了功能。

1.5.2 优化

编译器使用一套采用多种先进技术的高级优化，将 C 源代码编译为高效而紧凑的代码。优化包括适用于所有 C 代码的高级优化，以及利用 dsPIC DSC 架构特征专门针对 dsPIC DSC 器件的优化。

1.5.3 ANSI 标准函数库支持

MPLAB C30 带有一个完整的 ANSI 标准函数库。所有这些库函数都经过验证并遵循 ANSI C 库标准。这个函数库包括字符串处理、动态存储器分配、数据转换、计时和数学函数 （三角、指数和双曲线函数）。还包括用于文件处理的标准 I/O 函数，支持通过命令行模拟器对主机文件系统进行完全的访问。同时还提供低级文件 I/O 函数的完整功能性源代码，这可以作为需要这些功能的应用的一个起点。

1.5.4 灵活的存储器模型

编译器可同时支持大小两种代码和数据模型。小代码模型利用调用和分支指令的更有效形式，而小数据模型支持使用压缩指令对 SFR 空间的数据进行访问。

编译器支持两种模型来访问常量。“constants in data”模型使用数据存储器，由运行时库初始化。“constants in code”模型使用程序存储器，通过程序空间可视性（Program Space Visibility， PSV）窗口访问。

1.5.5 编译器驱动程序

MPLAB C30 包括一个强大的命令行驱动程序。通过这个驱动程序，应用程序的编译、汇编和链接可以一步完成 （参见图 1-1）。

注：

第 2 章 MPLAB C30 与 ANSI C 的差别

2.1 简介

本章讨论 MPLAB C30 语法支持的 C 语言和 1989 年标准 ANSI C 之间的差别。

2.2 主要内容

本章包括以下主要内容：

- 关键字差别
- 语句差别
- 表达式差别

2.3 关键字差别

本节说明 ANSI C 和 MPLAB C30 所接受 C 在关键字方面的差别。新关键字是基本 GCC 实现的一部分，本节的讨论基于标准的 GCC 文档，选择 GCC MPLAB C30 部分的特定语法和语义来讲述。

- 指定变量的属性
- 指定函数的属性
- 内联函数
- 指定寄存器中的变量
- 复数
- 双字整型
- 用 `typeof` 引用类型

2.3.1 指定变量的属性

MPLAB C30 的关键字 __attribute__ 用来指定变量或结构位域的特殊属性。关键字后的双括弧中的内容是属性说明。下面是目前支持的变量属性：

- address (addr)
- aligned (alignment)
- deprecated
- far
- mode (mode)
- near
- noload
- packed
- persistent
- reverse (alignment)
- section ("section-name")
- sfr (address)
- space (space)
- transparent_union
- unordered
- unused
- weak

也可以通过在关键字前后使用 __ （双下划线）来指定属性 （例如，用 __aligned__ 代替 aligned）。这样将使你在头文件中使用它们时不必考虑会出现与宏同名的情况。

要指定多个属性，可在双括弧内用逗号将属性分隔开，例如：

```
__attribute__ ((aligned (16), packed))。
```

> **注：** 一个项目中对变量属性的使用要一致，这很重要。例如，如果在文件 A 中用 far 属性定义了一个变量，在文件 B 中将其声明为 extern 而不带 far，就可能导致链接错误。

address (*addr*)

address 属性为变量指定绝对地址。这个属性不能与 section 属性同时使用；address 属性优先。带 address 属性的变量不能存放到 auto_psv 空间 （参见 space() 属性或 -mconst-in-code 选项）；这样做会产生警告，且编译器将此变量存放到 PSV 空间。

如果要将变量存放到 PSV 段，地址应为程序存储器地址。

```
int var __attribute__ ((address(0x800)));
```


aligned (*alignment*)

该属性为变量指定最小的对齐方式，用字节表示。对齐方式必须是 2 的次幂。例如，下面的声明：

```
int x __attribute__ ((aligned (16))) = 0;
```

使编译器按照 16 字节分配全局变量 x。对于 dsPIC DSC 器件，这可以与访问需要对齐的 操作数的 DSP 指令和寻址模式的 asm 语句配合使用。

在前面的例子中，我们可以显式地指定我们希望编译器对给定变量使用的对齐方式（用字节表示）。或者，我们可以省略对齐方式，而要求编译器为变量使用 dsPIC DSC 器件的最大有用对齐。例如，我们可以这样写：

```
short array[3] __attribute__ ((aligned));
```

当省略了对齐属性说明中的对齐方式时，编译器会自动将已声明变量的对齐方式设置为目标单片机任何数据类型所使用的最大对齐方式。在 dsPIC DSC 器件中，为双字节（1 个字）。

aligned 属性只能增大对齐；但可以通过指定 packed 属性来减小对齐 （见下文）。aligned 属性与 reverse 属性冲突，同时指定两者会产生错误。

deprecated

deprecated 属性使得包含这一属性的声明能被编译器特别识别到。当使用 deprecated 函数或变量时，编译器会发出警告。

deprecated 定义仍将被编译器执行，并被反映到目标文件中。例如，编译以下程序：

```
int __attribute__((__deprecated__)) i;
int main() {
  return i;
}
```

将产生警告：

```
deprecated.c:4: warning: `i' is deprecated (declared
 at deprecated.c:1)
```

在生成的目标文件中，仍以通常的方式定义了 i。

far

far 属性告知编译器不必将变量分配到 near（前 8 KB）数据空间中（即变量可以分配到数据存储器中的任何地址）。

mode (*mode*)

在变量声明中使用该属性来指定与模式 *mode* 对应的数据类型。实际上就是允许根据变量的宽度指定整数或浮点数类型。 *mode* 的有效值见下表：

模式	宽度	MPLAB® C30 类型
QI	8 位	`char`
HI	16 位	`int`
SI	32 位	`long`
DI	64 位	`long long`
SF	32 位	`float`
DF	64 位	`long double`

这一属性对于编写可在所有 MPLAB C30 支持的目标单片机之间移植的代码很有用。例如，如下函数将两个 32 位有符号整数相加，并返回一个 32 位有符号整数结果：

```
typedef int __attribute__((__mode__(SI))) int32;
int32
add32(int32 a, int32 b)
  {
        return(a+b);
  }
```

可以指定 `byte` 或 `__byte__` 模式指明模式对应于单字节整数， `word` 或 `__word__` 模式对应于单字整数， `pointer` 或 `__pointer__` 模式用于表示指针。

near

`near` 属性告知编译器变量将变量分配到 near 数据空间（数据存储器的前 8 KB）。对这种变量的访问有时比访问未分配 （或不知已分配）到 near 数据空间的变量效率高。

```
int num __attribute__ ((near));
```

noload

`noload` 属性指明应该为变量分配空间，但不应为变量装入初值。这一属性对于设计在运行时将变量装入存储器 （如从串行 EEPROM）的应用程序可能有用。

```
int table1[50] __ attribute__  ((noload)) = { 0 };
```


packed

packed 属性指定变量或结构位域采用最小的可能对齐方式 —— 变量占一个字节，位域占一位，除非用 aligned 属性指定了一个更大的值。

下面的结构中位域 x 被压缩，所以它紧接在 a 之后：

```
struct foo
{
 char a;
 int x[2] __ attribute__ ((packed));
};
```

> **注：** dsPIC 器件要求字按偶数字节对齐，因此在使用 packed 属性时要特别小心，避免运行时寻址错误。

persistent

persistent 属性指定在启动时变量不应被初始化或清零。具有 persistent 属性的变量可用于存储器件复位后仍保持有效的状态信息。

```
int last_mode __attribute__ ((persistent));
```

reverse (*alignment*)

reverse 属性为变量的结束地址加 1 指定最小对齐。对齐以字节指定，必须是 2 的次幂。反向对齐的变量可用于递减 dsPIC DSC 汇编语言中的模缓冲区。如果应用程序需要在 C 中定义的变量可从汇编语言访问，这一属性可能有用。

```
int buf1[128] __attribute__ ((reverse(256)));
```

reverse属性与aligned和section属性冲突。试图为反向对齐的变量指定一个段将被忽略，并发出警告。为同一个变量同时指定 reverse 和 aligned 会产生错误。带有 reverse 属性的变量不能存放到 auto_psv 空间 （参见 space 属性或 -mconst-in-code 选项）；试图这样做将导致警告，且编译器会将变量存放到 PSV 空间。

section ("*section-name*")

默认情况下，编译器将其生成的目标代码存放在 .data 和 .bss 段中。 section 属性允许指定变量 （或函数）存放到特定的段中。

```
struct array {int i[32];}
struct array buf __attribute__ ((section("userdata"))) = {0};
```

section 属性与 address 和 reverse 属性冲突。在这两种冲突情形下，段名将被忽略，并发出警告。这一属性还可能与 space 属性冲突。更多信息，参见关于 space 属性的说明。

sfr (*address*)

sfr 属性告知编译器变量是一个特殊功能寄存器 （SFR），同时使用 address 参数指定变量的运行时地址。

```
extern volatile int __attribute__ ((sfr(0x200)))u1mod;
```

为避免产生错误，需要使用 extern 说明符。

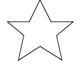
> **注：** 按照约定，仅在处理器头文件中使用 sfr 属性。为将一个普通变量定义到指定的地址，要使用 address 属性，且用 near 或 far 来指定正确的寻址模式。

space (*space*)

一般来说，编译器在一般数据空间内分配变量。可使用 space 属性来指示编译器将变量分配到特定存储空间。关于存储空间的更多论述参见**第 4.6 节 "存储空间"**。
space 属性接受如下参数：

data

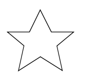
将变量分配到一般数据空间。可使用一般的 C 语句访问一般数据空间中的变量。这是默认的分配。



xmemory ——仅适用于 dsPIC30F/33F DSC

将变量分配到 X 数据空间。可使用一般的 C 语句访问 X 数据空间中的变量。xmemory 空间分配举例如下：

```
int x[32] __attribute__ ((space(xmemory)));
```



ymemory——仅适用于 dsPIC30F/33F DSC

将变量分配到 Y 数据空间。可使用一般的 C 语句访问 Y 数据空间中的变量。ymemory 空间分配举例如下：

```
int y[32] __attribute__ ((space(ymemory)));
```

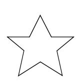
prog

将变量分配到程序空间中为可执行代码指定的段。程序空间中的变量不能使用一般的 C 语句访问，这些变量必须由编程人员显式访问，通常通过表访问行内汇编指令，或使用程序空间可视性窗口访问。

auto_psv

将变量分配到程序空间中为自动程序空间可视性窗口访问指定的编译器管理段。auto_psv 空间中的变量可使用一般的 C 语句来读（但不能写），且变量的分配空间最大为 32K 。当指定 space(auto_psv) 时，不能使用 section 属性指定段名；任何段名将被忽略并产生警告。 auto_psv 空间中的变量不能存放到特定地址或反对齐。

> **注：** 在启动时分配到 auto_psv 段中的变量不装入数据存储器。这一属性对于减少 RAM 使用可能有用。

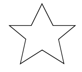


dma——仅适用于 PIC24H MCU 和 dsPIC33F DSC

将变量分配到 DMA 存储区。可以通过一般的 C 语句和 DMA 外设访问 DMA 存储区中的变量。可使用 `__builtin_dmaoffset()`（参见 *《dsPIC® 语言工具库》*（DS51456B_CN））来得到用于配置 DMA 外设的正确偏移量。

psv

将变量分配到程序空间中为程序空间可视性窗口访问指定的段。链接器将定位段，因此可以通过 PSVPAG 寄存器的设置来访问整个变量。 PSV 空间中的变量不是由编译器管理的，不能使用一般的 C 语句访问。这些变量必须由编程人员显式访问，通常使用表访问行内汇编指令，或使用程序空间可视性窗口访问。



eedata——仅适用于 dsPIC30F DSC

将变量分配到 EEData 空间。 EEData 空间中的变量不能使用一般的 C 语句访问。这些变量必须由编程人员显式访问，通常使用表访问行内汇编指令，或使用程序空间可视性窗口访问。

transparent_union

这是属于 `union` 型函数参数的属性，即相应的参数可以是任何联合成员的类型，但以第一个联合成员的类型传递参数。使用 transparent 联合的第一个成员的调用约定将参数传递给函数，而不是使用联合本身的调用约定。联合的所有成员必须具有相同的机器码表示，这对于保证参数传递正常进行是必需的。

unordered

`unordered` 属性表明变量存放的地址可以相对于所在 C 源文件中其他变量的位置而改变。

```
const int __attribute__ ((unordered)) i;
```

unused

这一变量属性表明变量可能不被使用。 MPLAB C30 不会为这种变量产生未使用变量警告。

weak

`weak` 属性声明 weak 符号。weak 符号可能被全局定义取代。当对外部符号的引用使用 `weak` 时，则链接时不需要该符号。例如：

```
extern int __attribute__((__weak__)) s;
int foo() {
  if (&s) return s;
  return 0; /* possibly some other value */
}
```

在上面的程序中，如果 `s` 没有被其他模块定义，程序仍会链接，但不会给 `s` 分配地址。若条件验证 `s` 已被定义，就返回它的值（如果它有值的话）。否则将返回 “`0`” 值。这个特征很有用，主要用于提供与任意库链接的通用代码。

weak 属性可以应用于函数和变量：

```
extern int __attribute__((__weak__)) compress_data(void *buf);
int process(void *buf) {
  if (compress_data) {
    if (compress_data(buf) == -1) /* error */
  }
  /* process buf */
}
```

在上述代码中，函数 compress_data 只有在与其他模块链接时才使用。是否使用该特性是由链接时决定的，而不是由编译时决定的。

weak 属性对定义的影响更为复杂，需要多个文件加以说明：

```
/* weak1.c */
int __attribute__((__weak__)) i;

void foo() {

  i = 1;
}

/* weak2.c */
int i;

extern void foo(void);

void bar() {
  i = 2;
}

main() {
  foo();
  bar();
}
```

在 weak2.c 中对 i 的定义使符号成为强定义。链接时不会出现错误，两个 i 指向同一个存储位置。为 weak1.c 中的 i 分配存储空间，但这个空间不可访问。

不能保证两个程序里的 i 具有相同的类型，如果将 weak2.c 中的 i 改为 float 型，仍然允许链接，但是函数 foo 的操作将无法预料。 foo 将向 32 位浮点值的最低有效部分写入一个值。相反，在 weak1.c 中把 i 的 weak 定义改为 float 型，将导致灾难性结果。这样会把一个 32 位的浮点值写到 16 位的整型地址中，覆盖掉紧接 i 之后存储的任何变量。

在只存在 weak 定义的情况下，链接器才选择为第一个这种定义分配存储空间。其他定义是不可访问的。

无论符号属于什么类型，操作是相同的；函数和变量具有相同的操作。

2.3.2 指定函数的属性

在 MPLAB C30 中，可以对程序中调用的函数进行某些声明，帮助编译器优化函数调用，且更准确地检查代码。

关键字 _attribute_ 允许在声明时指定特殊的属性。关键字后面紧跟双括弧中的属性说明。目前支持函数的下列属性：

- address (addr)
- alias ("target")
- const
- deprecated
- far
- format (archetype, string-index, first-to-check)
- format_arg (string-index)
- interrupt [([save(list)] [, irq(irqid)] [, altirq(altirqid)] [, preprologue(asm)])]
- near
- no_instrument_function
- noload
- noreturn
- section ("section-name")
- shadow
- unused
- weak

我们也可以通过在关键字前后使用 __ （双下划线）来指定属性 （例如，用 __shadow __ 代替 shadow）。这样使得在头文件中使用它们时不必考虑会出现与宏同名的情况。

我们要想在声明中指定多个属性，可以在双括弧内使用逗号将属性分隔开，或者在一个属性声明后紧跟另一个属性声明。

address (*addr*)

address 属性为函数指定绝对地址。这个属性不能与 section 属性同时使用；address 属性优先。

```
void foo() __attribute__ ((address(0x100))) {
...
}
```

alias (*"target"*)

alias 属性为另一个符号声明一个别名，必须指定这个符号。

使用这一属性会产生对对象的外部引用，必须在链接时解析该引用。

const

许多函数除了检查自身的参数外不会检查任何其他值，只会影响其返回值。可像算术运算符一样，对这种函数进行公共子表达式删除和循环优化。这些函数应该用属性 const 来声明。例如：

```
int square (int) __attribute__ ((const int));
```

也就是说，上述假设的 square 函数的实际被调用次数即使比程序指定的次数少一些也是安全的。

应该注意，如果函数有指针参数，且检查指针指向的数据，那么这种函数一定不能用 const 声明。同样，调用非 const 函数的函数通常也不能声明为 const。具有 void 返回值类型的 const 函数没有什么意义。

deprecated

关于 deprecated 属性的信息，请参阅**第 2.3.1 节 "指定变量的属性"**。

far

far 属性告知编译器不应该用更有效的调用指令形式来调用该函数。

format (*archetype*, *string-index*, *first-to-check*)

format 属性指定一个函数具有 printf、scanf 或 strftime 类型参数，要割据格式字符串检查这些参数的类型。例如，考虑以下声明：

```
extern int
my_printf (void *my_object, const char *my_format, ...)
     __attribute__ ((format (printf, 2, 3)));
```

以上语句使编译器检查对 my_printf 调用中的参数，确定是否与 printf 类型的格式字符串参数 my_format 一致。

参数 archetype 确定如何解释格式字符串，应该为 printf、 scanf 或 strftime 之一。参数 string-index 指定哪个参数是格式字符串参数 （参数从左至右编号，从 1 开始）， first-to-check 指定根据格式字符串检查的第一个参数的编号。对于不能检查参数的函数 （如 vprintf），指定第三个参数为 0。这种情况下，编译器仅检查格式字符串的一致性。

在上面的例子中，格式字符串 （my_format）是函数 my_print 的第二个参数，从第三个参数开始检查，所以 format 属性的正确参数是 2 和 3。

format 属性允许识别以格式字符串作为参数的用户自定义函数，所以 MPLAB C30 可以检查对这些函数的调用有无错误。每当要求这种警告 （使用 -Wformat）时，编译器总会检查 ANSI 库函数 printf、 fprintf、 sprintf、 scanf、 fscanf、sscanf、 strftime、 vprintf、 vfprintf 和 vsprintf 的格式，所以不必修改头文件 stdio.h。

format_arg (*string-index*)

format_arg 属性指定一个函数具有 printf 或者 scanf 类型的参数，修改这个函数（如将它翻译为另外一种语言），并把函数的结果传递给 printf 或 scanf 类型的函数。例如，考虑以下声明：

```
extern char *
my_dgettext (char *my_domain, const char *my_format)
      __attribute__ ((format_arg (2)));
```

上述语句使编译器检查对函数 my_dgettext 的调用中的参数，该函数的结果传递给 printf、 scanf 或 strftime 类型函数，确定是否与 printf 类型的格式字符串参数 my_format 一致。

参数 string-index 指定哪个参数是格式字符串参数（从 1 开始）。

format-arg 属性允许识别修改格式字符串的用户定义函数，所以 MPLAB C30 可以检查对 printf、 scanf 或 strftime 函数的调用，这些函数的操作数是对用户定义函数的调用。

interrupt [([save(*list*)] [, irq(*irqid*)] [, altirq(*altirqid*)] [, preprologue(*asm*)])]

使用这个选项来指明指定的函数是中断服务程序。当指定这个属性时，编译器将生成适用于中断服务程序的函数 prologue 和 epilogue 序列。可选的参数 save 指定函数 prologue和epilogue中分别保存和恢复的变量列表。可选参数 irq和altirq指定要使用的中断向量表 ID。可选参数 preprologue 指定要在编译器生成的 prologue 代码前生成的汇编代码。完整的说明和示例，请参阅**第 7 章 “中断”**。

near

near 属性告知编译器可以使用 call 指令的更有效形式调用函数。

no_instrument_function

如果指定命令行选项 -finstrument-functions，那么几乎所有用户函数的入口和出口处在编译时都会被插入 profiling 函数。而函数被指定此选项时将不执行上述操作。

noload

noload 属性指明应该为函数分配空间，但不应把实际代码装入存储器。如果应用程序设计为在运行时将函数装入存储器（如从 EEPROM），这一属性很有用。

```
void bar() __attribute__ ((noload)) {
...
}
```

noreturn

一些标准库函数是不能返回的，例如 abort 和 exit。 MPLAB C30 自动清楚这种情况。有些程序自定义了不会返回的函数，我们可以将这些函数声明为 noreturn 来告知编译器这种情况。

```
void fatal (int i) __attribute__ ((noreturn));

void
fatal (int i)
{
  /* Print error message. */
  exit (1);
}
```

noreturn 关键字告知编译器 fatal 不会返回。这可以优化代码，而不必考虑如果 fatal 返回会怎样。这可以在某种程度上优化代码。而且这样有助于避免未初始化变量的假警告。

对于 noreturn 函数，非 void 的返回值类型并没有什么意义。

section ("*section-name*")

通常，编译器将生成的代码存放在 .text 段中。但有时可能需要其他的段，或者需要将某些函数存放在特殊的段中。 Section 属性指定将一个函数存放在特定的段中。例如下面的声明：

```
extern void foobar (void) __attribute__ ((section (".libtext")));
```

上述语句将函数 foobar 存放在 .libtext 段中。

section 属性与 address 属性有冲突。忽略段名会导致警告。

shadow

shadow 属性使编译器使用影子寄存器而不是软件堆栈来保存寄存器。该属性通常与 interrupt 属性同时使用。

```
void __attribute__ ((interrupt, shadow)) _T1Interrupt (void)
```

unused

这个函数属性，表明函数可能不会被使用。 MPLAB C30 不会为这种函数发出未使用函数的警告。

weak

关于 weak 属性，参见**第 2.3.1 节 “指定变量的属性”**。

2.3.3 内联函数

通过声明一个函数为 inline，可以指示 MPLAB C30 将这个函数的代码集成到调用函数的代码中。通常这样可避免函数调用的开销，使代码执行速度更快。另外，若任何实际的参数值为常数，它们的已知值可允许在编译时进行简化，这样不用包含所有的内联函数代码。对代码量的影响是不容易预估的。使用内联函数，机器代码量视具体情况可能更大也有可能更小。

> **注：** 仅当函数定义可见（不只是有函数原型）时，才能使用函数内联。为将一个函数内联到多个源文件中，可将函数定义放在每个源文件包含的头文件中。

为将函数声明为内联，在其声明中使用 inline 关键字，例如：

```
inline int
inc (int *a)
{
  (*a)++;
}
```

（如果使用 -traditional 选项或 -ansi 选项，用 __inline__ 代替 inline。）还可以通过使用命令行选项 -finline-functions 将所有 “足够简单”的函数内联。编译器可以根据对函数大小的估计，直观地决定哪些函数足够简单，可以这样集成。

> **注：** 仅当使能 -finline 或优化时才识别 inline 关键字。

函数定义中的某些用法可能使函数不适合于内联替代。这些用法包括：varargs 的使用、 alloca 的使用，长度可变数据的使用，以及相对 goto 和非局部 goto 的使用。如果使用了命令行选项 -winline，当标识为 inline 的函数不能被替代时，会发出警告，并给出失败原因。

在 MPLAB C30 语法中，关键字 inline 不会影响函数的链接。

当一个函数同时为 inline 和 static 时，如果对该函数的所有调用都集成到调用函数中，且从不使用该函数的地址，那么该函数自身的汇编程序代码从不会被引用。在这种情况下， MPLAB C30 实际上并不输出该函数的汇编代码，除非指定命令行选项 -fkeep-inline-functions。有些调用由于各种原因不能被集成（特别是在函数定义之前的调用不能被集成，定义内的递归调用也不能被集成）。如果存在非集成的调用，那么会以通常方式将函数编译成汇编代码。如果程序引用函数的地址，也必须以通常的方式编译函数，因为它不能被内联。仅在内联函数被声明为 static，且函数定义在函数使用之前的情况下，编译器才会删除内联函数。

当 inline 函数不是 static 时，编译器必须假定其他源文件可能调用这个函数。因为全局符号只能在所有程序中定义一次，不能在其他源文件中定义该函数，所以其他源文件中的调用不能被集成。因此，非 static 的内联函数总是以通常的方式编译。

如果我们在函数定义中同时指定 inline 和 extern，这样定义的函数就只能用来内联。不能以通常的方式编译函数，即使显式地引用其地址。这种地址变成了一个外部引用，如同我们只是声明了函数却没有定义它。

同时使用 inline 和 extern 对于宏有类似的影响。使用这些关键字将一个函数的定义放在头文件中，并且将定义的另外一份拷贝 （不带 inline 和 extern）放在库文件中。头文件中的定义使得对于该函数的大多数调用被内联。如果还有任何使用该函数的地方，将引用库文件中的拷贝。

2.3.4 指定寄存器中的变量

MPLAB C30 允许把几个全局变量存放到指定的硬件寄存器中。

注：	使用太多的寄存器，尤其是寄存器 W0，可能影响 MPLAB C30 的编译能力。

我们也可以指定在其中存放普通寄存器变量的寄存器。

- 全局寄存器变量在整个程序执行过程中保留寄存器的值。这在程序中可能很有用，如编程语言解释程序，带有几个经常被访问的全局变量。
- 特定寄存器中的局部寄存器变量并不保留寄存器的值。编译器的数据流分析可以确定何时指定寄存器包含有效的值，何时可将指定寄存器用于其他用途。局部寄存器变量不使用时其中存储的值可被删除。对局部寄存器变量的引用可以被删除、移动或简化。

如果要将汇编指令的一个输出直接写到某个特定的寄存器，这些局部变量有时便于扩展行内汇编使用 （参见**第 8 章 “汇编语言和 C 模块混合编程”**）。（只要指定的寄存器符合为行内汇编语句中的操作数指定的约束就可以。）

2.3.4.1 定义全局寄存器变量

在 MPLAB C30 中，可通过以下语句来定义一个全局寄存器变量：

```
register int *foo asm ("w8");
```

其中， w8 是要使用的寄存器名。选择一个可被函数调用正常保存和恢复的寄存器（W8-W13），这样库函数就不会破坏它的值。

将一个全局寄存器变量定义到某个寄存器中，可完全保留该寄存器的值，至少在当前的编译中。在当前的编译中，寄存器不会被分配给函数中的任何其他用途。寄存器不会被这些函数保存和恢复。即使该寄存器的内容不被使用，也不会被删除，但是对该寄存器的引用可被删除、移动或简化。

从信号处理程序或者从多个控制线程访问全局寄存器变量是不安全的，因为系统库函数可能临时使用寄存器做别的工作 （除非你特别为待处理任务重新编译它们）。

同样不安全的是，使用一个全局寄存器变量的函数，通过函数 lose 来调用另外一个这样的函数 foo，而编译函数 lose 时未知该全局变量 （即在未声明该变量的源文件中）。这是因为 lose 可能会将其他某个值保存到该寄存器中。例如，不能在比较函数中使用传递给 qsort 的全局寄存器变量，因为 qsort 可能已经把其他值存放到该寄存器中了。用相同的全局寄存器变量定义来重新编译 qsort，可以避免此问题。

如果想重新编译实际上没有使用该全局寄存器变量的 qsort 或其他源文件，因此这些源文件不会将该寄存器用于其他用途，那么指定编译器命令行选项 -ffixed-reg 就足够了。这种情况下实际上不需要在其源代码中加一个全局寄存器声明。

一个函数若可能改变一个全局寄存器变量的值，它就不能安全地被不保存和恢复该变量编译的函数调用，因为这可能破坏调用函数返回时期望找到的值。因此，若一个程序片段使用了全局寄存器变量，作为该程序片段入口的函数必须显式地保存和恢复属于其调用函数的值。

库函数 longjmp 将恢复每个全局寄存器变量在 setjmp 时的值。

所有全局寄存器变量的声明必须在所有函数定义之前。如果这种声明在函数定义之后，寄存器可能被声明之前的函数用于其他用途。

全局寄存器变量不能有初值，因为可执行文件不能为一个寄存器提供初值。

2.3.4.2 为局部变量指定寄存器

可以通过以下语句用一个指定的寄存器定义局部寄存器变量：

```
register int *foo asm ("w8");
```

其中， w8 是使用的寄存器名。应该注意这与定义全局寄存器变量的语法相同，但是对于局部变量，这种定义应该出现在一个函数中。

定义这种寄存器不保留寄存器的值，流控制确定变量的值无效时，其他用途仍可使用这种寄存器。使用这一功能，可能使编译某些函数时可用寄存器太少。

该选项并不能保证 MPLAB C30 生成的代码始终将这一变量存放在指定的寄存器中。不可以在 asm 语句中，编写对该寄存器的显式引用，并假定它总是引用这个变量。

局部寄存器变量不使用时其分配可被删除。对局部寄存器变量的引用可以被删除、移动或简化。

2.3.5 复数

MPLAB C30 支持复数数据类型。我们可以用关键字 __complex__ 来声明整型复数和浮点型复数。

例如， __complex__ float x; 定义 x 为实部和虚部都是浮点型的变量。 __complex__ short int y; 定义 y 的实部和虚部都是 short int 型的。

要写一个复数数据类型的常量，使用后缀 “i” 或 “j”（两者之一，两者是等同的）。例如， 2.5fi 是 __complex__ float 型的， 3i 是 __complex__ int 型的。这种常量只有虚部值，但是我们可以通过将其与实常数相加来形成任何复数值。

要提取复数值符号 *exp* 的实部，写 `__real__` *exp*。类似地，用 `__imag__` 来提取虚部。例如：

```
__complex__ float z;
float r;
float i;

r = __real__ z;
i = __imag__ z;
```

当对复数型值使用算子 “~” 时，执行复数的共扼。

MPLAB C30 可以采用非邻近的方式分配复数自动变量，甚至可以将实部分配到寄存器中，而将虚部分配到堆栈中，反之亦然。调试信息格式无法表示这种非邻近的分配，所以 MPLAB C30 把非邻近的复数变量描述为两个独立的非复数类型变量。如果实际变量名是 `foo`，那么两个假设变量命名为 `foo$real` 和 `foo$imag`。

2.3.6 双字整型

MPLAB C30 支持长度为 `long int` 两倍的整型数据类型。对于有符号整型，写 `long long int`，而对于无符号整型，使用 `unsigned long long int`。可以通过在整型上添加后缀 `LL` 得到 `long long int` 类型的整型常量，在整数上添加后缀 `ULL` 得到 `unsigned long long int` 类型的整型常量。

可以在算术运算中像使用其他整型一样使用这些类型。这些数据类型的加、减和位逻辑布尔运算是开放源代码的，但是，这些数据类型的除法与移位不是开放源代码的。这些不开放源代码的运算要使用 MPLAB C30 自带的特殊库函数。

2.3.7 用 typeof 引用类型

引用表达式类型的另一种方法是使用 `typeof` 关键字。使用这个关键字的语法与 `sizeof` 相似，但是其结构在语义上类似于用 `typedef` 定义的类型名。

有两种方法写 `typeof` 的参数：使用表达式或者使用类型。以下为使用表达式的例子：

```
typeof (x[0](1))
```

这里假设 `x` 是函数数组，描述的类型就是函数值的类型。

以下为使用类型名作为参数的例子：

```
typeof (int *)
```

这里，描述的类型是指向 `int` 的指针。

如果写一个包含在 ANSI C 程序中时必须有效的头文件，要使用 `__typeof__`，而不要使用 `typeof`。

typeof 结构可用于可使用 typedef 名的任何地方。例如，可以在声明和强制类型转换中，或者 sizeof 或 typeof 的内部使用它。

- 用 x 指向的类型声明 y：
  `typeof (*x) y;`
- 将 y 声明为这种值的数组：
  `typeof (*x) y[4];`
- 将 y 声明为指向字符的指针数组：
  `typeof (typeof (char *)[4]) y;`
  它等同于如下的传统 C 声明：
  `char *y[4];`

为了弄清楚 typeof 声明的含义，以及为什么是有用的方法，我们用以下宏改写它：

```
#define pointer(T) typeof(T *)
#define array(T, N) typeof(T [N])
```

现在声明可以这样改写：

```
array (pointer (char), 4) y;
```

这样， array (pointer (char), 4) 是指向 char 的四个指针的数组类型。

2.4 语句差别

本节讲述普通 ANSI C 与 MPLAB C30 所接受 C 之间的语句差别。语句差别是基本 GCC 实现的一部分，本节讨论的内容基于标准 GCC 文档，选择了 GCC 中 MPLAB C30 部分的特定语法和语义来讲述。

- 将标号作为值
- 省略操作数的条件表达式
- case 范围

2.4.1 将标号作为值

可以用单目运算符 “&&” 获得在当前函数 （或包含函数）中定义的标号的地址。值的类型为 void *。这个值为常量，并可在这种类型的常量有效的任何地方使用这个值。例如：

```
void *ptr;
...
ptr = &&foo;
```

为使用这些值，需要能跳转到值。这通过计算 goto 语句 goto *exp; 来实现。例如：

```
goto *ptr;
```

可使用 void * 类型的任何表达式。

这些常量的一个用途是用于初始化用作跳转表的静态数组：

```
static void *array[] = { &&foo, &&bar, &&hack };
```

然后就可以通过索引来这样选择标号：

```
goto *array[i];
```

注：	这并不检查下标是否超出范围 （C 中的数组索引从不这样做）。

这种标号值数组的用途与 switch 语句很类似。 switch 语句更整齐，比数组更好。

标号值的另外一个用途是在线程代码的解释程序中。解释程序函数中的标号可存储在线程代码中用于快速调度。

这种机制可能被错误使用，而跳转到其他函数的代码中。编译器不能阻止这种现象的发生，因此必须小心，确保目标地址对于当前函数有效。

2.4.2 省略操作数的条件表达式

条件表达式的中间操作数可以被省略。如果第一个操作数非零，它的值就是条件表达式的值。

因此，对于表达式：

```
x ? : y
```

如果 x 的值非零，表达式的值就是 x 的值；否则，就是 y 的值。

这个例子完全等价于：

```
x ? x : y
```

在这个简单的例子中，省略中间操作数并不是特别有用。当第一个操作数存在或者可能存在 （如果它是一个宏参数）副作用时，省略中间操作数就变得特别有用。那么重复中间操作数将产生副作用两次。省略中间操作数使用了已经计算过的值，而不会因为重新计算而产生不希望的影响。

2.4.3 case 范围

可以如下在单个 case 标号中指定一个连续值的范围：

`case` *`low`* `...` *`high`*`:`

这与各个 case 标号的适当数字有相同的作用， 每个数字对应从 *low* 到 *high* 中的每个整数值。

这一功能对于 ASCII 字符码范围特别有用：

```
case 'A' ... 'Z':
```

注意：在 ... 两边要写空格，否则它和整数一起使用时可能出现解析错误。例如要这样写：

```
case 1 ... 5:
```

而不要这样写：

```
case 1...5:
```

2.5 表达式差别

本节讨论普通 ANSI C 和 MPLAB C30 所接受的 C 之间的表达式差别。

2.5.1 二进制常量

前面有 0b 或 0B 的一串二进制数字 （数字 “0”后跟字母 “b”或 “B”）视为二进制整型。二进制数字由数字 “0”和 “1”组成。例如，十进制数字 255 可用二进制表示为 0b11111111。

像其他整型常量一样，二进制常量可以以字母 “u”或 “U”为后缀来指定为无符号型。二进制常量也可以以字母 “l”或 “L”为后缀，指定为长整型。类似地，后缀 “ll”或 “LL”表示双字整型的二进制常量。

第 3 章 使用 MPLAB C30 C 编译器

3.1 简介

本章讨论通过命令行使用 MPLAB® C30 C 编译器。关于在 MPLAB IDE 中使用 MPLAB C30 的信息，请参阅 *《dsPIC® DSC 语言工具入门》*（DS70094C_CN）。

3.2 主要内容

本章介绍以下内容：

- 概述
- 文件命名约定
- 选项
- 环境变量
- 预定义常量
- 通过命令行编译单个文件
- 通过命令行编译多个文件

3.3 概述

编译驱动程序（`pic30-gcc`）对 C 和汇编语言模块及库文件进行编译、汇编和链接。大多数编译器命令行选项对于 GCC 工具集的所有实现都是通用的。只有少数是专门针对 MPLAB C30 编译器的。

编译器命令行的基本形式如下：

```
pic30-gcc [options] files
```

注：	命令行选项和文件扩展名要区分大小写。

在**第 3.5 节 “选项”**中对可用的选项进行了描述。

例如，下面的命令行编译、汇编和链接 C 源文件 `hello.c`，生成可执行文件 `hello.exe`。

```
pic30-gcc -o hello.exe hello.c
```

3.4 文件命名约定

编译驱动程序识别如下文件扩展名，文件扩展名要区分大小写。

表 3-1： 文件名

扩展名	定义
file.c	必须预处理的 C 源文件。
file.h	头文件 （不对其进行编译或链接）。
file.i	不应预处理的源文件。
file.o	目标文件。
file.p	预过程抽象汇编语言文件。
file.s	汇编代码。
file.S	必须预处理的汇编代码。
其他	要传递给链接器的文件。

3.5 选项

MPLAB C30 提供了许多控制编译的选项，它们都是区分大小写的。

- 针对 dsPIC DSC 器件的选项
- 控制输出类型的选项
- 控制 C 语言的选项
- 警告与错误控制选项
- 调试选项
- 控制优化的选项
- 控 制预处理器的选项
- 汇编选项
- 链接选项
- 目录搜索选项
- 代码生成约定选项

3.5.1 针对 dsPIC DSC 器件的选项

关于存储模型的更多信息，请参阅**第 4.7 节 “存储模型”**。

表 3-2： 针对 dsPIC® DSC 器件的选项

选项	定义
`-mconst-in-code`	将常量存放在 auto_psv 空间中。编译器将使用 PSV 窗口访问这些变量。（**这是默认设置。**）
`-mconst-in-data`	将常量存放到数据存储空间中。
`-merrata=` *`id[,id]*`*	此选项使能特定的勘误变通解决方案 （errata workaround），由 *`id`* 标识。 *`id`* 的有效值时常改变，对于某个特定的器件可能不需要。`list` 的 *`id`* 将列出目前支持的勘误表标识符以及对勘误表的简单描述。 `all` 的 *`id`* 将使能所有目前支持的勘误变通解决方案。
`-mlarge-code`	使用大代码模型编译。对于被调用函数是局部函数还是全局函数不做假设。 选择这个选项时，大于 32k 的单个函数是不支持的，这样的函数可能导致汇编时错误，因为函数内部的所有分支都是短跳转形式 .
`-mlarge-data`	使用大代码模型编译。不假定静态变量和外部变量的位置。
`-mcpu=` `<it>target`	此选项选择具体的目标器件 （且如果要进行这一处理的话，将目标器件选择信息传递给汇编器和链接器）。这个选项会对某些预定义常量的设置产生影响，更多信息参见**第 3.7 节 “预定义常量”**。可在本编译器发布版本附带的 README.TXT 中找到所接受目标器件的完整列表。
 `-mpa`[1]	使能过程抽象优化。对嵌套深度没有限制。
 `-mpa=`*`n`*[1]	允许过程抽象优化达到 *n* 级。如果 *n* 为 0，那么禁止优化。如果 *n* 为 1，允许一级抽象；也就是说，源代码中的指令序列可以抽象为子程序。如果 *n* 为 2，允许二级抽象；也就是说，在一级抽象中包含在子程序中的指令可以抽象为更深一级的子程序。对于更大的 *n* 值，继续依此类推。 实际上是为了将子程序调用嵌套的深度限制为最大值 *n*。
 `-mno-pa`[1]	不允许过程抽象优化。 （**这是默认设置。**）

注 **1**：过程抽象的操作与内联函数相反。这一过程设计为通过翻译单元从多处抽取相同的代码序列，并存放到一个公共代码区。尽管这个选项一般并不会提高所生成代码的运行时性能，却可以显著减小代码长度。采用 `-mpa` 编译的程序可能难以调试；在使用 COFF 目标格式调试时，不推荐使用这个选项。

过程抽象是生成汇编文件后，一个独立的编译阶段。这个阶段不跨翻译单元优化。当使能过程优化阶段时，行内汇编代码仅限于有效的机器指令。不能使用无效的机器指令或指令序列，或汇编伪指令 （段伪指令、宏和包含文件等），否则过程抽象阶段会失败，影响输出文件的生成。

表 3-2: 针对 dsPIC® DSC 器件的选项 （续）

选项	定义
-mno-isr-warn	默认情况下，如果在识别到的中断向量名后没有加 __interrupt__，编译器将产生警告。此选项将禁止这个功能。
-momf=*omf*	选择编译器使用的 OMF （目标模块格式）。 *omf* 说明符可以为下列之一： coff 生成 COFF 目标文件。（**这是默认设置。**） elf 生成 ELF 目标文件。 ELF 目标文件使用的调试格式为 DWARF 2.0。
-msmall-code	使用小代码模型编译。假定被调用函数在调用函数的 32K 字内。（**这是默认设置。**）
-msmall-data	使用小数据模型编译。假定所有静态变量和外部变量位于数据存储空间的低 8KB 地址。（**这是默认设置。**）
-msmall-scalar	与 -msmall-data 类似，不同的是仅假定静态标量和外部标量位于数据存储空间的低 8KB 地址。（**这是默认设置。**）
-mtext=*name*	指定 -mtext=name 将文本 （程序代码）放入名为 *name* 的段中，而不是默认的 .text 段中。等号两边不能有空格。
-msmart-io [=0|1|2]	该选项试图对传递给 printf 和 scanf 函数，以及这两个函数 “f” 和 “v”形式的格式字符串进行静态分析。非浮点型参数的使用将转换成使用仅支持整型的库函数形式。 -msmart-io=0 将禁止这个选项，而 -msmart-io=2 将使编译器转换带有变量或未知格式参数的函数调用。**默认情况下** -msmart-io=1，将仅转换它能验证的立即数值。

注 1： 过程抽象的操作与内联函数相反。这一过程设计为通过翻译单元从多处抽取相同的代码序列，并存放到一个公共代码区。尽管这个选项一般并不会提高所生成代码的运行时性能，却可以显著减小代码长度。采用 -mpa 编译的程序可能难以调试；在使用 COFF 目标格式调试时，不推荐使用这个选项。

过程抽象是生成汇编文件后，一个独立的编译阶段。这个阶段不跨翻译单元优化。当使能过程优化阶段时，行内汇编代码仅限于有效的机器指令。不能使用无效的机器指令或指令序列，或汇编伪指令 （段伪指令、宏和包含文件等），否则过程抽象阶段会失败，影响输出文件的生成。

3.5.2 控制输出类型的选项

表 3-3： 输出类型控制选项

选项	定义
`-c`	编译或汇编源文件，但不链接。默认的文件扩展名为 `.o`。
`-E`	在预处理过程之后，即正常运行编译器之前停止。默认输出文件为 `stdout`。
`-o` *`file`*	将输出放在 *`file`* 中。
`-S`	在正常编译之后，即调用汇编器之前停止。默认输出文件扩展名为 `.s`。
`-v`	在编译的每个阶段打印执行的命令。

`-x`	可用 `-x` 选项显式地指定输入语言：
	`-x` *`language`*
	为后面的输入文件显式地指定语言 （而不是让编译器根据文件名后缀选择默认的语言）。这个选项适用于其后直到下一个 `-x` 选项之前的所有输入文件。 MPLAB C30 支持下面的值：
	`c  c-header  cpp-output`
	`assembler  assembler-with-cpp`
	
	`-x none`
	关闭所有语言指定，随后的文件按其文件名后缀处理。如果已使用另一个 `-x` 选项，这是默认但必需的。例如：
	`pic30-gcc -x assembler foo.asm bar.asm -x none main.c mabonga.s`
	
	没有 `-x none` 时，编译器将假定所有输入文件都为汇编语言。

`--help`	打印命令行选项的描述。

3.5.3 控制 C 语言的选项

表 3-4： C 语言控制选项

选项	定义
-ansi	支持 （且仅支持）所有 ANSI 标准的 C 程序。
-aux-info filename	对于在翻译单元中声明和 / 或定义的函数，包括头文件中的函数，输出到给定文件名的原型声明中。除了 C，这个选项在其他语言中通常被忽略。除了声明以外，文件在注释中指出了每个声明的来源 （源文件和行），不论声明是隐含的，原型的，还是非原型的 （在行号和冒号后面的第一个字符中，I、N 代表新的，O 代表旧的），也不论它来自声明还是定义 （在随后的字符中，分别用 C 和 F 代表）。如果是函数定义，在函数声明之后的注释中，还提供 K&R 型参数列表，后跟这些参数的声明。
-ffreestanding	指明编译在独立环境中进行。这意指 -fno-builtin 选项。独立的环境就是其中可能不存在标准库，程序也不必在主函数中启动的环境。最显而易见的例子就是 OS 内核。这与 -fno-hosted 等价。
-fno-asm	不识别 asm、 inline 或 typeof 关键字，因此代码可以将这些单词用作标识符。可以使用关键字 __asm__、__inline__ 和 __typeof__ 。-ansi 意指 -fno-asm。
-fno-builtin -fno-builtin-*function*	不识别不以 __builtin_ 作为前缀开始的内建函数。
-fsigned-char	使 char 型变量为有符号，就像 signed char。**（这是默认设置。）**
-fsigned-bitfields -funsigned-bitfields -fno-signed-bitfields -fno-unsigned-bitfields	如果声明时未使用 signed 或 unsigned，这些选项用来控制位域是有符号还是无符号的。默认情况下，这样的位域都是有符号的，除非使用 -traditional，它使位域总是无符号的。
-funsigned-char	使 char 型变量无符号，就像 usigned char。
-fwritable-strings	将字符串存储到可写的数据段中，但不要使字符串成为唯一的。

3.5.4 警告与错误控制选项

警告是诊断消息，它报告非本质错误、但有危险的语法结构，或暗示可能存在错误。

可以使用以 -W 开头的选项请求许多特定的警告，例如，使用 -Wimplicit 请求关于隐式声明的警告。每条这些特定的警告选项也可以用以 -Wno- 开头的相反形式来关闭警告，如 -Wno-implicit。本手册只列出了这两种形式中的一种，这两种形式都不是默认的。

下面的选项控制 MPLAB C30 C 编译器产生的警告的数量和种类。

表 3-5： **-WALL 隐含的警告 / 错误选项**

选项	定义
-fsyntax-only	检查代码的语法，除此之外不做任何事情。
-pedantic	发出 严格 ANSI C 要求的所有警告；拒绝所有使用禁止扩展名的程序。
-pedantic-errors	类似于 -pedantic，只是发出错误而不是警告。
-w	禁止所有警告消息。
-Wall	使能本表中列出的所有 -W 选项。这将使能关于某些用户认为有问题的，及容易避免的 （或修改来禁止警告的）语法结构的所有警告，即使是与宏一起。
-Wchar-subscripts	如果数组下标具有 char 类型则警告。
-Wcomment -Wcomments	当注释开始符号 /* 出现在 /* 注释中，或反斜杠换行出现在 // 注释中发出警告。
-Wdiv-by-zero	编译时发现整数除以 0 则警告。禁止这个消息可使用 -Wno-div-by-zero。浮点数除以 0 不会警告，因为它可以是获得无穷大和 NaN 的一种合法方法。（这是默认情况。）
-Werror-implicit-function-declaration	函数在声明前被使用将给出错误。
-Wformat	检查对 printf 和 scanf 等函数的调用，确保所提供参数的类型与指定的格式字符串相符合。
-Wimplicit	等价于同时指定 -Wimplicit-int 和 -Wimplicit-function-declaration。
-Wimplicit-function-declaration	函数在声明前被使用将给出警告。
-Wimplicit-int	如果声明没有指定类型则警告。
-Wmain	如果 main 的类型有问题则警告。 main 应该是一个具有外部链接的函数，它返回 int，并带有正确类型的 0、 2 或 3 个参数。

-Wmissing-braces	如果一个聚集或联合的初始化中括号不全则警告。在下面的例子中， a 的初始化中括号不全，而对 b 的初始化是正确的。
	`int a[2][2] = { 0, 1, 2, 3 };`
	`int b[2][2] = { { 0, 1 }, { 2, 3 } };`

表 3-5: **-WALL** 隐含的警告 / 错误选项 （续）

选项	定义

`-Wmultichar`
`-Wno-multichar`
	使用多字符的 `character` 常量时警告。通常出现这样的常量是由于输入错误。由于这种常量具有实现定义的值，不应将它们用在可移植代码中。下面举例说明了多字符 `character` 常量的使用：
	`char`
	`xx(void)`
	`{`
	`return('xx');`
	`}`

`-Wparentheses`	在某些上下文中省略圆括号时警告，如在需要真值的上下文中有一个赋值，或者运算符嵌套的运算优先级容易混淆时。
`-Wreturn-type`	当函数定义为其返回值类型默认为 `int` 时发出警告。如果函数的返回值类型不是 `void`，那么不带返回值的任何 `return` 语句都会导致产生警告。
`-Wsequence-point`	由于违背 C 标准中的顺序点规则而导致代码中有未定义的语义时发出警告。 C 标准定义了 C 程序中根据顺序点对表达式求值的顺序，顺序点代表程序各部分执行的局部顺序：在顺序点之前执行的部分和顺序点之后执行的部分。这些在一个完整表达式（不是一个更大的表达式的一部分）的求值之后，在对第一个运算符（&&、 ||、 ? : 或 , （逗号）运算符）求值之后，在调用函数前（但在对其参数和表示被调用函数的表达式求值后），以及某些其他地方发生。除了顺序点规则指定的顺序外，未指定表达式的子表达式的求值顺序。所有这些规则仅规定了局部的顺序，而没有规定全局的顺序，因为，如在一个表达式中调用了两个函数，而它们之间没有顺序点，就没有指定函数调用的顺序。但是，标准委员会规定函数调用不能重叠。 没有指定在顺序点之间，对对象的值的修改何时生效。操作依赖于这一点的程序有不确定的操作； C 标准规定，“在上一个顺序点和下一个顺序点之间，对象所储存的值最多只能被表达式求值修改一次。而且，前一个值是只读的以便确定将被储存的值。”如果程序违反这些规则，任何特定实现的结果都是完全不可预估的。 具有未定义操作的代码示例有：`a = a++;`，`a[n] = b[n++];` 及 `a[i++] = i;`。这个选项不能诊断某些更复杂的情况，并可能给出偶然错误的结果，但通常在检测程序中的这类问题时，这个选项还是很有效的。

表 3-5： **-WALL** 隐含的警告 / 错误选项 （续）

选项	定义
-Wswitch	每当 switch 语句中有一个枚举类型的索引，并且这个枚举的一个或多个指定码缺少 case 时发出警告。（默认标号的存在禁止这个警告。）当使用这个选项时，枚举范围之外的 case 标号也会引起警告。
-Wsystem-headers	打印关于系统头文件中语法结构的警告消息。系统头文件的警告通常是被禁止的，因为通常认为它们不会有真正的问题，只会使编译器的输出可读性更差。使用这个命令行选项告知 MPLAB C30 发出关于系统头文件的警告，就像在用户代码中一样。但是，注意将 -Wall 与该选项一起使用时不会对系统头文件中的未知 pragma 伪指令发出警告，这时，必须同时使用 -Wunknown-pragmas。
-Wtrigraphs	遇到三字母组合时发出警告 （假定使能了三字母组合）。
-Wuninitialized	使用自动变量而没有先对其初始化时发出警告。 这些警告只有在允许优化时才出现，因为它们需要只有优化时才计算的数据流信息。 仅当将变量分配给寄存器时才产生这些警告。因此，对于声明为 volatile 的变量，或是变量地址被占用，或者大小不是 1， 2， 4 和 8 字节的变量不会产生这些警告。同样对于结构、联合或数组，即使它们在寄存器中，也不会产生这些警告。 注意，当一个变量只是用于计算一个值而变量本身不会被使用时，也不会产生警告，因为在警告被打印前，这样的计算就会被数据流分析删除。
-Wunknown-pragmas	当遇到一个 MPLAB C30 无法理解的 #pragma 伪指令时发出警告。如果使用这个选项，甚至对系统头文件中的未知 pragma 伪指令也会发出警告。如果警告只能通过命令行选项 -Wall 来使能，情况就不是这样了。
-Wunused	每当变量除了其声明外未被使用过时，每当函数声明为 static 但从未定义时，每当声明了标号但未使用时，每当一条语句的计算结果未被显式使用时，发出警告。要获得未使用的函数参数的警告，必须同时指定 -W 和 -Wunused。 强制转换表达式类型可以避免禁止对表达式的这种警告。同样地， unused 属性可以禁止对未使用的变量、参数和标号的警告。
-Wunused-function	每当声明了 static 函数但没有定义函数时，或一个非内联 static 函数未使用时，发出警告。
-Wunused-label	声明了一个标号但未使用时发出警告。要禁止这种警告，可以使用 unused 属性 （参见第 **2.3.1 节 “指定变量的属性”**）。

表 3-5： **-WALL** 隐含的警告 / 错误选项 （续）

选项	定义
-Wunused-parameter	当对函数参数进行了声明但从未使用时，发出警告。要禁止这种警告，使用 unused 属性 （参见**第 2.3.1 节 “指定变量的属性”**）。
-Wunused-variable	当对局部变量或非常量的 static 变量进行了声明但从未使用时，发出警告。要禁止这种警告，使用 unused 属性 （参见**第 2.3.1 节 “指定变量的属性”**）。
-Wunused-value	语句的计算结果未显式使用时发出警告。要禁止这种警告，可以将表达式类型转换为 void。

下面是不被 -Wall 隐含的 -W 选项。其中有些是关于用户通常认为不会有问题，但有时会希望检查一下的语法结构的警告。其他是在某些情况下必须或很难避免的语法结构的警告，没有简单的方法来修改代码以禁止这些警告。

表 3-6： **-WALL** 不隐含的警告 / 错误选项

选项	定义
-W	为以下事件输出额外警告消息： • 非易变的自动变量可能会被对 longjmp 的调用改变。这些警告仅在优化编译时才会出现。编译器仅识别对 setjmp 的调用，而不会知道将在何处调用 longjmp，信号处理程序可以在代码中的任何地方调用 longjmp。因此，即使当实际没有问题时也可能会产生警告，因为实际上不能在会产生问题的地方调用 longjmp。 • 函数可以通过 return value; 和 return; 退出。函数体结束时不传递任何返回值的语句视为 return;。 • 表达式语句或者逗号表达式的左侧没有副作用。为了禁止这种警告，将未使用表达式的类型强制转换为 void。例如，表达式 x[i,j] 会产生警告，而表达式 x[(void)i,j] 不会产生警告。 • 用 < 或 <= 将无符号值与 0 比较。 • 出现了像 x<=y<=z 这样的不等式；这等价于 (x<=y ? 1 : 0) <= z，这只是普通数学表示的不同解释罢了。 • 存储类型修饰符如 static 在声明中没有放在最前面，根据标准，这种用法已经过时了。 • 如果还指定了 -Wall 或 -Wunused，会出现关于未使用变量的警告。 • 当将有符号值转换为无符号值时，比较有符号值和无符号值会产生不正确的结果。（但是如果还指定了 -Wno-sign-compare 的话，就不会产生警告。）

表 3-6： **-WALL** 不隐含的警告 / 错误选项 （续）

选项	定义

-W	• 聚集的初始化中括号不全。例如，下面的代码由于在初始化 x.h 时漏掉了括号会产生警告 :
	`struct s { int f, g; };`
	`struct t { struct s h; int i; };`
	`struct t x = { 1, 2, 3 };`
	• 聚集的初始化中没有初始化所有成员，例如，下面的代码由于 x.h 会被隐式初始化为零而会产生警告：
	`struct s { int f, g, h; };`
	`struct s x = { 3, 4 };`

-Waggregate-return	定义或调用了返回结构或联合的任何函数时产生警告。
-Wbad-function-cast	当将函数调用强制转换为不匹配类型时产生警告。例如，如果 int foof() 被强制转换为任何 * 指针类型。
-Wcast-align	当强制转换指针类型，使目标所需分配的存储空间增加时产生警告。例如，如果将 char * 强制转换为 int * 会产生警告。
-Wcast-qual	当对指针进行强制类型转换，从目标类型中去掉类型限定符时，会产生警告。例如，将 const char * 强制转换为普通的 char * 就会产生警告。
-Wconversion	如果一个原型导致一个参数的类型转换与没有原型时不同，则发出警告。这包括定点型转换为浮点型或反之，及改变定点参数符号或宽度的转换，与默认的提升相同时除外。 当负的整型常量表达式隐式转换为无符号类型时也发出警告。例如，如果 x 为无符号类型，赋值 x = -1 将产生警告。但是，显式的强制类型转换，如 (unsigned) -1，不会产生警告。
-Werror	使所有警告变为错误。
-Winline	一个函数已声明为内联，或指定了 -finline-functions 选项时，如果函数不能被内联，将产生警告。
-Wlarger-than-*len*	当定义了大于 *len* 字节的对象时产生警告。
-Wlong-long -Wno-long-long	使用 long long 类型时发出警告。这是默认设置。为禁止警告消息，使用 -Wno-long-long。仅当使用 -pedantic 标志时，才考虑标志 -Wlong-long 和 -Wno-long-long。
-Wmissing-declara-tions	如果在定义一个全局函数之前没有先对其进行声明将产生警告。即使定义本身提供了原型，也要在定义全局函数之前先声明它。
-Wmissing-format-attribute	如果使能了 -Wformat，可指定 format 属性的函数也会产生警告。注意这些函数仅是可指定 format 属性的函数，不是已指定 format 属性的函数。如果不使能 -Wformat，这一选项不起作用。

表 3-6： **-WALL** 不隐含的警告 / 错误选项 （续）

选项	定义
`-Wmissing-noreturn`	对可指定 `noreturn` 属性的函数产生警告。这些函数仅是可指定这一属性的函数，并不是已指定了这一属性的函数。手工检验这些函数时要小心。实际上，在添加 `noreturn` 属性之前也不要返回；否则可能会引入微小的代码生成错误。
`-Wmissing-prototypes`	如果全局函数在定义之前没有先声明原型会产生警告。即使定义本身提供了原型也会发出这个警告。（这个选项可用于检测不在头文件中声明的全局函数。）
`-Wnested-externs`	如果在函数内部遇到了 `extern` 声明，发出警告。
`-Wno-deprecated-declarations`	不要对使用通过 `deprecated` 属性指定为 deprecated 的函数、变量和类型发出警告。
`-Wpadded`	如果一个结构中包含了填充，不管是为了对齐结构的一个元素，还是为了对齐整个结构，都发出警告。
`-Wpointer-arith`	对于与函数类型或 `void` 的长度有关的任何类型发出警告。为方便用 `void *` 指针和指向函数的指针计算， MPLAB C30 将这些类型的长度分配为 1。
`-Wredundant-decls`	如果在同一个作用域内多次声明了任何符号则发出警告，即使多个声明都有效且没有改变任何符号。
`-Wshadow`	当一个局部变量屏蔽另一个局部变量时发出警告。
`-Wsign-compare` `-Wno-sign-compare`	当比较有符号值和无符号值时，将有符号值转换为无符号值，比较产生不正确结果时发出警告。这个警告也可通过 `-W` 来使能；要获得 `-W` 的其他警告，而不获得这个警告，使用 `-W -Wno-sign-compare`。
`-Wstrict-prototypes`	如果对一个函数的定义或声明没有指定参数类型则发出警告。（如果函数定义或声明前有指定函数参数类型的声明，则允许旧式函数定义而不发出警告。）
`-Wtraditional`	如果某些语法结构在传统 C 和 ANSI C 中操作不同，产生警告。 • 宏参数出现在宏体中的字符串常量中。在传统 C 中，这些宏参数将替代参数，但在 ANSI C 中是常量的一部分。 • 在一个块中声明为 external 的函数，在块结束后被使用。 • switch 语句有 `long` 类型的操作数。 • 非静态函数声明后跟一个静态函数声明。某些传统 C 编译器不接受这种语法结构。
`-Wundef`	如果在 `#if` 伪指令中对一个未定义的标识符求值会产生警告。
`-Wunreachable-code`	如果编译器检测到代码将永远不会被执行到则发出警告。即使在有些情况下，受影响的代码行的一部分能被执行到，这个选项也可能产生警告，因此在删除明显执行不到的代码时要小心。例如，函数被内联时，警告可能表明仅在函数的一个内联拷贝中，该行执行不到。

表 3-6： **-WALL** 不隐含的警告 / 错误选项 （续）

选项	定义
`-Wwrite-strings`	字符串常量类型为 `const char[length]` 时，将一个字符串常量的地址复制到一个非常量 `char *` 指针会产生警告。这些警告有助于在编译时查找试图写字符串常量的代码，但仅是在声明和原型中使用 `const` 时非常小心的前提下。否则，这是不安全的，这也是 `-Wall` 为什么不要求这些警告的原因。

3.5.5 调试选项

表 3-7： 调试选项

选项	定义
`-g`	产生调试信息。 MPLAB® C30 支持同时使用 `-g` 和 `-O`，因此可以调试优化的代码。调试优化代码的缺点是有时可能产生异常结果： - 某些声明的变量可能根本不存在； - 控制流程可能短暂异常转移； - 某些语句可能由于计算常量结果或已经获得其值而不执行； - 某些语句可能由于被移出循环在不同的地方执行。 尽管如此，证明还是可以调试优化输出的。这使优化可能有错误的程序变得合理。
`-Q`	使编译器打印它在编译的每个函数名，并在结束时打印关于每遍编译的一些统计信息。
`-save-temps`	不要删除中间文件。将中间文件放在当前目录中，并根据源文件命名它们。因此，用 “`-c -save-temps`” 编译 “`foo.c`” 将生成下面的文件： ‘`foo.i`’ （预处理文件） ‘`foo.p`’ （预过程抽象汇编语言文件） ‘`foo.s`’ （汇编语言文件） ‘`foo.o`’ （目标文件）

3.5.6 控制优化的选项

表 3-8： 一般优化选项





选项	定义
`-O0`	不要优化 （这是默认设置）。 不指定 `-O` 选项，编译器的目标是降低编译成本，使调试产生期望的结果。语句是独立的：如果在语句中插入断点暂停程序，然后可以给任何一个变量赋一个新的值或将程序计数器更改到指向函数中的任何其他语句，得到希望从源代码得到的结果。 编译器仅将声明为 `register` 的变量分配到寄存器中。
`-O` `-O1`	优化。优化编译需要花费更多的时间，且对于较大的函数，需要占用更多的存储空间。 指定 `-O` 选项时，编译器试图减小代码尺寸并缩短执行时间。 指定 `-O` 选项时，编译器开启 `-fthread-jumps` 和 `-fdefer-pop`，并开启 `-fomit-frame-pointer`。
`-O2`	执行更多优化。 MPLAB® C30 几乎执行所有支持的优化，而不进行空间和速度的权衡。 `-O2` 选项使能除循环展开（`-funroll-loops`）、函数内联（`-finline-functions`）及严格别名优化（`-fstrict-aliasing`）之外的所有可选优化。这个选项还使能强制复制存储器操作数 （`-fforce-mem`）及帧指针删除（`-fomit-frame-pointer`）。与 `-O` 相比，这个选项增加了编译时间，但提高了生成代码的性能。
`-O3`	执行最多的优化。 `-O3` 开启所有 `-O2` 指定的优化并开启内联函数选项。
`-Os`	优化代码尺寸。 `-Os` 使能一般不增加代码尺寸的所有 `-O2` 优化。同时执行用于减小代码尺寸的其他优化。

下面的选项控制特定的优化。 -O2 选项启用这些优化中除 -funroll-loops、-funroll-all-loops 和 -fstrict-aliasing 外的所有优化选项。

在少数情况下，当需要进行 “微调”优化时，可以使用下面的选项。

表 3-9： 特定的优化选项

选项	定义
 -falign-functions -falign-functions=*n*	将函数的开头对齐到下一个大于 *n* 的 2 的次幂，最多跳过 *n* 字节。*例如，* -falign-functions=32 将函数对齐到下一个 32 字节边界，但是 -falign-functions=24 仅在可以通过跳过等于或小于 23 字节能对齐到下一个 32 字节边界的情况下，才将函数对齐到下一个 32 字节边界。 -fno-align-functions 和 -falign-functions=1 是等价的，表明函数不会被对齐。 汇编器仅当 *n* 为 2 的次幂时，才支持这个标志；因此 n 是向上舍入的。如果不指定 *n*，则使用由机器决定的默认设置。
-falign-labels -falign-labels=*n*	将所有分支的目标地址对齐到 2 的次幂边界，像 -falign-functions 一样，最多跳过 *n* 字节。这个选项可能容易使代码速度变慢，因为当以代码的通常流程到达分支的目标地址时，它必须插入空操作。 如果 -falign-loops 或 -falign-jumps 可用，并且大于这个值，则使用它们的值。 如果不指定 *n*，则使用由机器决定的默认设置，很可能是 1，表明不对齐。
 -falign-loops -falign-loops=*n*	将循环对齐到 2 的次幂边界，像 -falign-functions 一样，最多跳过 *n* 字节。希望循环能执行许多次，从而补偿执行的任何空操作。 如果不指定 *n*，则使用由机器决定的默认设置。
-fcaller-saves	通过在函数调用前后发出其他指令来保护和恢复寄存器，使能将值分配到会被函数调用破坏的寄存器中。仅当这种分配能生成更好的代码时才进行这种分配。
 -fcse-follow-jumps	在公共子表达式消除中，当任何其他路径都不到达跳转的目标地址时，浏览跳转指令。例如，当 CSE 遇到一条带有 else 子句的 if 语句时，当条件检测为假时， CSE 将跟随跳转。
 -fcse-skip-blocks	这与 -fcse-follow-jumps 类似，但使 CSE 跟随根据条件跳过块的跳转。当 CSE 遇到一个没有 else 子句的简单 if 语句时，-fcse-skip-blocks 使 CSE 跟随 if 前后的跳转。
 -fexpensive-optimizations	执行许多成本较高的次要优化。
-ffunction-sections -fdata-sections	将每个函数或数据项存放到输出文件中其自己的段。函数名或数据项名决定输出文件中的段名。 仅当使用这些选项有明显的好处时，才使用这些选项。当指定这些选项时，汇编器和链接器可能生成较大的目标文件和可执行文件，且速度较慢。

表 3-9： 特定的优化选项 （续）

选项	定义
 -fgcse	执行全局公共子表达式消除。这会同时执行全局常量和复制传播。
-fgcse-lm	使能 -fgcse-lm 时，全局公共子表达式消除将试图移动仅能被向其中存储破坏的装载。这允许将包含装载 / 存储序列的循环改变为循环外的装载，以及循环内的复制 / 装载。
-fgcse-sm	当使能 -fgcse-sm 时，将在公共子表达式消除后运行存储移动。这试图将存储移出循环。当将这个选项与 -fgcse-lm 一起使用时，包含装载 / 存储序列的循环可改变为循环前的装载和循环后的存储。
-fmove-all-movables	强制将循环内所有不可变的计算移出循环。
-fno-defer-pop	每次函数调用时，总是在函数一返回时就弹出函数的参数。编译器通常允许几个函数调用的参数累积在堆栈中，并将所有参数一次弹出堆栈。
 -fno-peephole -fno-peephole2	禁止特定于机器的窥孔 （peephole）优化。窥孔优化发生在编译过程中的不同点。-fno-peephole 禁止对机器指令进行窥孔优化，而 -fno-peephole2 禁止高级窥孔优化。要完全禁止窥孔优化，要同时使用这两个选项。
 -foptimize-register-move -fregmove	试图重新分配 move 指令中的寄存器编号，并作为其他简单指令的操作数来增加关联的寄存器数量。 -fregmove 和 -foptimize-register-moves 是相同的优化。
-freduce-all-givs	强制循环中的所有一般归纳变量降低强度。 这些选项可能生成更好或更差的代码；其结果在很大程度上取决于源代码中循环的结构。
 -frename-registers	试图通过使用寄存器分配后余下的寄存器来避免经过调度的代码中的假相关性。这种优化对于有许多寄存器的处理器比较有用。但它可能使调试无法进行，因为变量将不会存储在固定的寄存器中。
 -frerun-cse-after-loop	在执行循环优化后，重新运行公共子表达式消除。
-frerun-loop-opt	运行循环优化两次。
 -fschedule-insns	试图对指令重新排序，以消除 dsPIC® DSC 写 - 后 - 读（Read-After-Write）停顿 （详情请参阅 《*dsPIC30F 系列参考手册*》（DS70046D_CN））。一般可提高性能，而不会影响代码长度。
 -fschedule-insns2	类似于 -fschedule-insns，但要求在进行寄存器分配后再执行一次指令调度。
-fstrength-reduce	执行降低循环强度和删除迭代变量优化。

表 **3-9**：　　特定的优化选项 （续）



<table>
<tr><th>选项</th><th>定义</th></tr>
<tr><td><code>-fstrict-aliasing</code></td><td>允许编译器采用适用于被编译语言的最严格别名规则。对于 C，这根据表达式的类型进行优化。尤其是，假定一种类型的对象不会和另一种类型的对象存放在同一地址，除非类型几乎相同。例如， <code>unsigned int</code> 可引用 <code>int</code>，但不能引用 <code>void*</code> 或 <code>double</code>。字符类型可引用任何其他类型。
特别要注意下面的代码：<pre>union a_union {
 int i;
 double d;
};

int f() {
 union a_union t;
 t.d = 3.0;
 return t.i;
}</pre>不读最后写入的联合成员，而读其他联合成员 （称为 “type-punning”）比较常见。即使对于 <code>-fstrict-aliasing</code>，如果通过联合类型访问存储器，type-punning 也是允许的。因此上面的代码可得到期望的结果。但下面的代码可能得不到期望的结果：<pre>int f() {
 a_union t;
 int* ip;
 t.d = 3.0;
 ip = &t.i;
 return *ip;
}</pre></td></tr>
<tr><td><code>-fthread-jumps</code></td><td>是一种优化，检测一个转移的目标语句是否包含另一个条件判断。如果是这样，第一个转移改变为指向第二个转移的目标语句，或紧随其后的语句，这取决于条件是真还是假。</td></tr>
<tr><td><code>-funroll-loops</code></td><td>执行循环展开优化。仅对在编译时或运行时其迭代次数可以确定的循环进行这种优化。 <code>-funroll-loops</code> 隐含了 <code>-fstrength-reduce</code> 和 <code>-frerun-cse-after-loop</code>。</td></tr>
<tr><td><code>-funroll-all-loops</code></td><td>执行循环展开优化。对于所有的循环执行这种优化，通常这种优化会使程序运行较慢。
<code>-funroll-all-loops</code> 隐含了 <code>-fstrength-reduce</code> 和 <code>-frerun-cse-after-loop</code>。</td></tr>
</table>

-*fflag* 形式的选项指定依赖于机器的标志。大多数标志有正的形式和负的形式；-ffoo 的负的形式为 -fno-foo。在下表中，仅列出了一种形式 （非默认的形式）。



表 3-10： 独立于机器的优化选项

选项	定义
-fforce-mem	在对存储器操作数进行算术运算之前，强制将存储器操作数复制到寄存器中。这样通过使所有存储器引用可能的公共子表达式，可生成更好的代码。当它们不是公共子表达式时，指令组合应该删除单独的寄存器装载。-O2 开启这个选项。
-finline-functions	将所有简单的函数合并到其调用函数中。编译器直观地决定哪些函数足够简单值得这样合并。如果合并了对某个给定函数的所有调用，且函数声明为 static，则通常该函数本身不作为汇编代码输出。
-finline-limit=*n*	默认情况下， MPLAB C30 限制可内联的函数的大小。这个选项允许控制显式声明为 inline 的函数 （即用 inline 关键字标记的函数）的这一限制。 *n* 是可内联的函数的大小，以虚拟指令的条数为单位 （参数处理不包括在内）。n 的默认值为 10000。增加这个值可能导致被内联的代码更多，并可能增加编译时间和存储器开销。 减小这个值通常使编译更快，更少的代码被内联 （可能程序执行速度变慢）。这一选项对于使用内联的程序尤其有用。 **注：**在这里，虚拟指令代表函数大小的抽象测量。它不代表汇编指令条数，同样对于不同版本的编译器，它的确切含义可能会有所不同。
-fkeep-inline-functions	即使合并了对一个给定函数的所有调用，且函数声明为 static，输出函数的一个独立的运行时可调用形式。这个选项不影响 extern 内联函数。
-fkeep-static-consts	当没有开启优化时，发出声明为 static const 的变量，即使变量没有被引用。 MPLAB C30 默认使能这个选项。如果需要强制编译器检查是否引用了这个变量，而不管是否开启了优化，使用 -fno-keep-static-consts 选项。
-fno-function-cse	不要将函数的地址存放在寄存器中；使调用 constant 函数的每条指令显式包含函数的地址。 这个选项导致生成的代码效率不高，但对于某些企图修改程序的人来说，会对不使用这个选项而生成的优化程序感到束手无策。

表 **3-10**： 独立于机器的优化选项 （续）

选项	定义
-fno-inline	不要理会 inline 关键字。这个选项通常用于使编译器不要展开任何内联函数。如果不使能优化，不会展开任何内联函数。
-fomit-frame-pointer	对于不需要帧指针的函数，不要将帧指针存放在寄存器中。这可以避免指令保护、设置和恢复帧指针；它还使一个额外的寄存器可用于许多函数。
-foptimize-sibling-calls	优化同属和尾递归调用。

3.5.7 控制预处理器的选项

表 **3-11**： 预处理器选项

选项	定义

-A*question* (*answer*)	断言问题 *question* 的答案 *answer*，以防用预处理条件，如 #if #*question*(*answer*) 来测试问题。 -A- 禁止通常描述目标机器的标准断言。
	例如， main 的函数原型可声明如下：
	`#if #environ(freestanding)`
	`int main(void);`
	`#else`
	`int main(int argc, char *argv[]);`
	`#endif`
	-A 命令行选项可用于在两个原型之间进行选择。例如，为选择二者中的第一个，可使用下面的命令行选项：
	-Aenviron(freestanding)

-A -*predicate* =*answer*	取消带谓词 *predicate* 和答案 *answer* 的断言。
-A *predicate* =*answer*	进行带谓词 *predicate* 和答案 *nswer* 的断言。这个形式比仍然支持的老形式 -A predicate(answer) 好，因为这个形式不使用 shell 特殊字符。
-C	告知预处理器不要舍弃注释。与 -E 选项一起使用。
-dD	告知预处理器不要按照正确的顺序将宏定义移动到输出中。
-D*macro*	将字符串 1 作为宏定义来定义宏 *macro*。
-D*macro*=*defn*	将宏 *macro* 定义为 *defn*。在任何 -U 选项之前处理命令行中的所有 -D 选项。
-dM	告知预处理器仅输出实际上处于预处理结尾的一系列宏定义。与 -E 选项一起使用。
-dN	与 -dD 类似，不同之处在于宏参数和内容被忽略。输出中仅包括 #define name。
-fno-show-column	不要在诊断中打印列号。如果诊断被不理解列号的程序 （如 dejagnu）浏览，这可能是必要的。
-H	打印使用的每个头文件的名字以及其他正常活动。

表 3-11： 预处理器选项 （续）

选项	定义
-I-	仅对于 #include "*file*"，才搜索在 -I- 选项之前 -I 选项指定的任何目录；对于 #include <*file*>，则不搜索这些目录。 如果在 -I- 之后用 -I 选项指定了另外的目录，则对于所有 #include 伪指令，都搜索这些目录。（一般情况下以这种方式使用所有 -I 目录。） 另外， -I- 选项禁止将当前目录 （即当前输入文件所在的目录） 作为 #include "*file*" 的第一个搜索目录。无法覆盖 -I- 的这个作用。通过 -I，可以指定搜索调用编译器时为当前目录的目录。这与预处理器在默认情况下的操作不完全相同，但一般情况下都可以这样做。 -I- 并不禁止使用头文件的标准系统目录。因此， -I- 和 -nostdinc 是独立的。
-I*dir*	将目录 *dir* 添加到要在其中搜索头文件的目录列表的开头。这可用于覆盖系统头文件，替代为您自己的版本，因为在搜索系统头文件目录之前搜索这些目录。如果使用多个 -I 选项，则以自左向右的顺序浏览目录，最后搜索标准系统目录。
-idirafter *dir*	将目录 *dir* 添加到辅助包含路径中。当一个头文件在主包含路径 （ -I 添加的路径）的任何目录中都找不到时，搜索辅助包含路径的目录。
-imacros *file*	在处理常规输入文件之前，将文件处理为输入，舍弃生成的输出。由于舍弃了由文件生成的输出， -imacros *file* 的唯一作用是使文件中定义的宏可用在主输入中。 命令行中的任何 -D 和 -U 选项始终在 -imacros *file* 之前处理，而与写这些选项的顺序无关。所有 -include 和 -imacros 选项以写这些选项时的顺序处理。
-include *file*	在处理常规输入文件之前，将文件处理为输入。实际上，首先编译文件的内容。命令行中的任何 -D 和 -U 选项始终在 -include *file* 之前处理，而与写这些选项的顺序无关。所有 -include 和 -imacros 选项以写这些选项时的顺序处理。
-iprefix *prefix*	指定 *prefix* 作为后面 -iwithprefix 选项的前缀。
-isystem *dir*	将一个目录添加到辅助包含路径的开头，将其标记为系统目录，因此可像处理标准系统目录一样处理这个目录。
-iwithprefix *dir*	将一个目录添加到辅助包含路径。目录名由前缀和 *dir* 组成，其中前缀由前面的 -iprefix 指定。如果没有指定前缀，将使用包含编译器安装路径的目录作为默认目录。
-iwithprefixbefore *dir*	将一个目录添加到主包含路径。目录名由前缀和 *dir* 组成，这与 -iwithprefix 相同。

表 3-11：　预处理器选项 （续）

选项	定义
`-M`	告知预处理器输出适合于描述每个目标文件的相关性的 make 的规则。对于每个源文件，预处理器输出目标为该源文件目标文件名且其相关性为它使用的所有 `#include` 头文件的 make 规则。这个规则可以为单行的或者太长时可用 \ 换行符来继续。规则列表打印在标准输出中，而不是打印在预处理的 C 程序中。 `-M` 隐含 `-E` （参见**第 3.5.2 节 “控制输出类型的选项”**）。
`-MD`	与 `-M` 类似，但将相关性信息写到一个文件，编译继续进行。包含相关性信息的文件的名字与带 `.d` 扩展名的源文件名字相同。
`-MF` *file*	当与 `-M` 或 `-MM` 一起使用时，指定在其中写入相关性信息的文件。如果不给定 `-MF` 开关，预处理器将发送规则到微处理器输出发送到的地方。 当与驱动程序选项 `-MD` 或 `-MMD` 一起使用时，`-MF` 覆盖默认的相关性输出文件。
`-MG`	将缺少的头文件视为生成的文件，并假定它们位于源文件所在的目录中。如果指定了 `-MG`，那么必须也指定 `-M` 或 `-MM`。 `-MD` 或 `-MMD` 不支持 `-MG`。
`-MM`	类似于 `-M`，但输出仅涉及到用 `#include` “*file*" 包含的用户头文件。用 `#include` <*file*> 包含的系统头文件被忽略。
`-MMD`	类似于 `-MD`，但仅涉及到用户头文件，不涉及到系统头文件。

`-MP`	这个选项指示 CPP 除主文件外，还要为每个相关性添加假目标，使每个不依赖于任何其他。如果删除头文件时不更新 Make-file 来匹配，这些假规则将避开 make 发出的错误。
	下面是典型的输出：
	`test.o: test.c test.h`
	`test.h:`

`-MQ`	与 `-MT` 相同，但它将特定于 Make 的任何字符用引号括起来。
	`-MQ '$(objpfx)foo.o'` 给出 `$$(objpfx)foo.o:`
	`foo.c`
	默认的目标自动被引号括起来，就像指定了 `-MQ` 一样。

`-MT` *target*	改变相关性生成发出的规则的目标。默认情况下， CPP 采用主输入文件的名字，包含任何路径，删除任何文件后缀 （如 `.c`），并添加平台的通常目标后缀。结果就是目标。
	`-MT` 选项将目标设置为你指定的字符串。如果需要多个目标，可件它们指定为 `-MT` 的一个参数，或使用多个 `-MT` 选项。
	例如：
	`-MT '$(objpfx)foo.o'` 可能得到 `$(objpfx)foo.o:`
	`foo.c`

表 3-11： 预处理器选项 （续）

选项	定义
-nostdinc	不要在标准系统目录中搜索头文件。仅搜索用 -I 选项指定的目录 （及当前目录，如果需要的话）。（关于 -I 选项的信息，请参阅**第 3.5.10 节 “目录搜索选项”**。） 通过同时使用 -nostdinc 和 -I-，可将头文件搜索路径限制为仅包括显式指定的目录。
-P	告知预处理器不要产生 #line 伪指令。与 -E 选项一起使用 （参阅**第 3.5.2 节 “控制输出类型的选项”**）。
-trigraphs	支持 ANSI C 三字母组合。-ansi 选项也有这个作用。
-U*macro*	取消宏 *macro* 定义。 -U 选项在所有 -D 选项之后，但在任何 -include 和 -imacros 选项之前起作用。
-undef	不要预定义任何非标准宏。 （包括结构标志。）

3.5.8 汇编选项

表 3-12： 汇编选项

选项	定义
-Wa,*option*	把 *option* 作为一个选项传递给汇编器。如果 *option* 中包含逗号，说明有多个选项通过逗号分隔开。

3.5.9 链接选项

如果使用了 -c、 -S 或 -E 选项中的任何一个，则链接器不会运行，且不应将目标文件名用作参数。

表 3-13： 链接选项

选项	定义
-L*dir*	将目录 *dir* 添加到命令行选项 -l 指定的在其中搜索库的目录列表中。
-l*library*	链接时搜索名为 *library* 的库。 链接器在标准目录列表中搜索库，实际上是一个名为 lib*library*.a 的文件。链接器随后对这个文件的使用，就好像已经通过文件名精确指定了这个文件一样。 在命令中的何处写这个选项是有所不同的，链接器按照指定库文件和目标文件的顺序来处理这些文件。因此， foo.o -lz bar.o 先搜索 foo.o，再搜索库 z，最后搜索 bar.o。如果 bar.o 引用 libz.a 中的函数，则可能不装载这些函数。 搜索的目录包括几个标准系统目录和使用 -L 指定的任何目录。 通常采用这种方法找到的文件是库文件 （其成员为目标文件的归档文件）。链接器通过浏览归档文件查找定义目前引用过但未定义的符号的成员来处理归档文件。但如果找到的文件是一个普通的目标文件，则以通常的方式链接这个文件。使用 -l 选项 （如 -lmylib）和指定文件名（如 libmylib.a）的唯一不同之处在于， -l 按照指定搜索几个目录。 默认情况下，链接器被指示在 <install-path>\lib 中搜索 -l 选项指定的库。对于安装到默认路径的编译器，这个目录为： c:\Program Files\Microchip\MPLAB C30\lib。 可使用在**第 3.6 节 "环境变量"**中定义的环境变量覆盖这个操作。
-nodefaultlibs	链接时不要使用标准系统库文件。仅指定的库文件会被传递给链接器。编译器可能产生对 memcmp、 memset 和 memcpy 的调用。这些入口通常由标准编译器库中的入口解析。当指定这个选项时，应通过其他某个机制来提供这些入口点。
-nostdlib	链接时不要使用标准系统启动文件或库文件。没有启动文件，仅指定的库文件会被传递给链接器。编译器可能产生对 memcmp、 memset 和 memcpy 的调用。这些入口通常由标准编译器库中的入口解析。当指定这个选项时，应通过其他某个机制来提供这些入口点。
-s	从可执行文件删除所有符号表和重定位信息。
-u *symbol*	假定 *symbol* 未定义，强制链接库模块来定义这个符号。可对不同的符号多次使用 -u 来强制装载其他库模块，这样做是合法的。
-Wl,*option*	将 *option* 作为一个选项传递给链接器。如果 *option* 包含逗号，它包含逗号分隔开的多个选项。
-Xlinker *option*	将 *option* 作为一个选项传递给链接器。可使用这个选项提供 MPLAB C30 不知如何识别的特定系统链接器选项。

3.5.10 目录搜索选项

表 3-14： 目录搜索选项

选项	定义
-B*prefix*	这个选项指定在哪里查找可执行文件、库文件、头文件和编译器本身的数据文件。 编译器驱动程序运行子程序 pic30-cpp、 pic30-cc1、pic30-as 和 pic30 中的一个或多个子程序。它将其运行的每个程序加上 *prefix* 作为前缀。 对于要运行的每个子程序，编译器驱动程序首先使用 -B 前缀 （如果存在的话）。如果找不到子程序，或未指定 -B，驱动程序将使用 PIC30_EXEC_PREFIX 环境变量 （如果设置了的话）中保存的值。更多信息，请参阅**第 3.6 节 "环境变量"**。最后，驱动程序将在当前的 PATH 环境变量中搜索子程序。 有效指定目录名的 -B 前缀也适用于链接器中的库，因为编译器将这些选项翻译为链接器的 -L 选项。它们也适用于预处理器中的头文件，因为编译器将这些选项转化为预处理器的 -isystem 选项。在这种情况下，编译器在前缀上附加 include。指定很像 -B 这样的前缀的另外一种方法是使用环境变量 PIC30_EXEC_PREFIX。
-specs=*file*	为覆盖当确定哪些开关传递给 pic30-cc1、 pic30-as 和 pic30-ld 等时， pic30-gcc 驱动程序使用的默认设置，在编译器读入标准 specs 文件后处理文件。可在命令行中指定多个 -specs=*file*，以自左向右的顺序处理这些文件。

3.5.11 代码生成约定选项

-f*flag* 形式的选项指定独立于机器的标志。大多数标志都有正的形式和负的形式；-ffoo 负的形式为 -fno-foo。在下表中，仅列出了一种形式 （非默认形式）。

表 3-15： 代码生成约定选项

选项	定义
-fargument-alias -fargument-noalias -fargument-noalias-global	指定参数之间以及参数和全局数据之间的可能关系。 -fargument-alias 指定实参 （形参）可互相引用，并可引用全局存储。 -fargument-noalias 指定实参不能互相引用，但可引用全局存储。 -fargument-noalias-global 指定实参不能互相引用，也不能引用全局存储。 每种语言都自动使用语言标准所要求的选项。不需要自己使用这些选项。
-fcall-saved-*reg*	将名为 *reg* 的寄存器视为函数保存的可分配寄存器。甚至可在其中分配临时变量或跨调用有效的变量。如果函数使用了寄存器 *reg*，那么采用这种方式编译的函数将保护和恢复这个寄存器。 对帧指针和堆栈指针使用这个标志是错误的。将这个标志用于在机器执行模型中有固定重要作用的其他寄存器，将产生灾难性结果。 将这个标志用于保存函数返回值的寄存器将产生另一种灾难性结果。 所有模块中对这个标志的使用应该一致。
-fcall-used-*reg*	将名为 *reg* 的寄存器视为被函数调用破坏的可分配寄存器。可将这个寄存器分配给临时变量或跨调用无效的变量。采用这个选项编译的函数不会保护和恢复寄存器 *reg*。 对帧指针或堆栈指针使用这个选项是错误的。将这个标志用于在机器执行模型中有固定重要作用的其他寄存器，将产生灾难性结果。 所有模块中对这个标志的使用应该一致。
-ffixed-*reg*	将名为 *reg* 的寄存器视为固定寄存器；生成的代码绝对不能引用它（除非作为堆栈指针、帧指针或某个其他固定的功能）。 *reg* 必须为寄存器的名字，如 -ffixed-w3。

-finstrument-functions	编译时在函数的入口和出口生成 instrumentation 调用。在函数入口之后和函数出口之前，将通过当前函数的地址及其调用地址来调用下面的 profiling 函数。
	`void __cyg_profile_func_enter`
	`(void *this_fn, void *call_site);`
	`void __cyg_profile_func_exit`
	`(void *this_fn, void *call_site);`
	第一个参数是当前函数的起始地址，可在符号表中查找到。
	profiling 函数应由用户提供。
	函数 instrumentation 要求使用帧指针。某些优化级别禁止使用帧指针。使用 -fno-omit-frame-pointer 将禁止这一点。

表 3-15： 代码生成约定选项 （续）

选项	定义
	instrumentation 也可用于在其他函数中扩展内联的函数。 profiling 调用表明从概念上来讲在哪里进入和退出内联函数。这意味着这种函数必须具有可寻址形式。如果对一个函数的所有使用都扩展内联，这会增加额外增加代码长度。如果要在 C 代码中使用 `extern inline`，必须提供这种函数的可寻址形式。 可对函数指定属性 `no_instrument_function`，在这种情况下不会进行 instrumentation。
`-fno-ident`	忽略 `#ident` 伪指令。
`-fpack-struct`	将所有结构成员无缝隙地压缩在一起。通常不希望使用这个选项，因为它使代码不是最优化的，且结构成员的偏移量与系统库不相符。 dsPIC® DSC 器件要求字按偶数字节边界对齐，因此当使用 packed 属性时要小心，避免运行时寻址错误。
`-fpcc-struct-return`	像长值一样，将短 `struct` 和 `union` 值返回到存储器中，而不是返回到寄存器中。这样做效率不高，但其优点是可以使 MPLAB® C30 编译的文件与其他编译器编译的文件兼容。 短结构和联合指长度和对齐都与整型匹配的结构和联合。
`-fno-short-double`	默认情况下，编译器使用与 `float` 等价的 `double` 型。这个选项使得 `double` 与 `long double` 等价。如果模块通过参数传递直接或通过共享缓冲空间间接共用 double 数据，跨模块混合使用这个选项可能会产生异常结果。无论使用哪个开关设置，随产品提供的库都可正常工作。
`-fshort-enums`	按照 `enum` 类型声明的可能值范围的需要，为其分配字节。具体来说， `enum` 类型等价于有足够空间的最小整型。
`-fverbose-asm` `-fno-verbose-asm`	在输出的汇编代码中加入额外的注释信息以增强可读性。 默认设置为 `-fno-verbose-asm`，将给出额外的信息，当比较两个汇编文件时有用。
`-fvolatile`	将通过指针进行的所有存储器引用视为 volatile。
`-fvolatile-global`	将对外部和全局数据项的所有存储器引用视为 volatile。使用这个开关对于 static 数据没有影响。
`-fvolatile-static`	将对 static 数据的所有存储器引用视为 volatile。

3.6 环境变量

本节中提到的变量是可选的，但是如果定义了这些变量，将由编译器使用。如果没有设置下面某些环境变量的值，编译器驱动程序或其他子程序，可能选择为这些变量确定适当的值。驱动程序或其他子程序，利用有关 MPLAB C30 安装的内部知识。只要安装结构是完整的，所有子目录和可执行文件在相同的相对路径中，驱动程序或子程序就能确定可使用的值。

表 3-16： 与编译器有关的环境变量

选项	定义
PIC30_C_INCLUDE_PATH	此变量的值是一个分号分隔开的目录列表，很像 PATH。当 MPLAB® C30 搜索头文件时，它在搜索标准头文件目录之前，搜索 -I 指定的目录之后，搜索此变量中列出的目录。 如果未定义该环境变量，预处理器根据标准安装选择适当的值。默认情况下，在下面的目录中搜索头文件： <install-path>\include 和 <install-path>\support\h。
PIC30_COMPILER_PATH	PIC30_COMPILER_PATH 的值是一个分号分隔的目录列表，很像 PATH。搜索子程序时，如果 MPLAB C30 使用 PIC30_EXEC_PREFIX 找不到子程序，它在这个变量指定的目录中搜索。
PIC30_EXEC_PREFIX	如果设置了 PIC30_EXEC_PREFIX，它指定要在编译器执行的子程序的名字中使用的前缀。当这个前缀和子程序名一起使用时，不添加目录分隔符，但如果需要的话，可以指定一个以斜杠符结束的前缀。如果 MPLAB C30 使用指定的前缀找不到子程序，它将在 PATH 环境变量中查找。 如果 PIC30_EXEC_PREFIX 环境变量未设置或设置为空值，则编译器驱动程序根据标准安装选择适当的值。如果安装没有被修改，那么驱动程序将能找到所需要的子程序。 使用 -B 命令行选项指定的其他前缀优先于 PIC30_EXEC_PREFIX 的用户定义值或驱动程序定义 值。 通常情况下，最好将此值保持为未定义，让驱动程序查找子程序。
PIC30_LIBRARY_PATH	这个变量的值是分号分隔的目录列表，很像 PATH。这个变量指定要传递给链接器的目录列表。驱动程序对这个变量的默认求值为： <install-path>\lib; <install-path>\support\gld.
PIC30_OMF	指定 MPLAB C30 要使用的 目标模块格式 （Object Module Format，OMF）。默认情况下，工具生成 COFF 目标文件。如果环境变量 PIC30_OMF 的值为 elf，工具将生成 ELF 目标文件。
TMPDIR	如果设置了 TMPDIR，它指定临时文件使用的目录。 MPLAB C30 使用临时文件来保存编译的一个阶段的输出，这个输出将用作编译的下一个阶段的输入：例如，预处理器的输出是编译器的输入。

3.7 预定义常量

可使用几个常量来定制编译器输出。

3.7.1 常量

下表中的预处理符号由所使用的编译器定义。

编译器	符号	是否通过 -ansi 命令行选项定义？
MPLAB® C30	C30	否
	__C30	是
	__C30__	是
ELF 版本	C30ELF	否
	__C30ELF	是
	__C30ELF__	是
COFF 版本	C30COFF	否
	__C30COFF	是
	__C30COFF__	是

下表中的符号定义目标器件系列。

符号	是否通过 -ansi 命令行选项定义？
__dsPIC30F__	是
__dsPIC33F__	是
__PIC24F__	是
__PIC24H__	是

此外，编译器还根据用 -mcpu= 设置的目标器件定义符号。例如， -mcpu=30F6014 定义了符号 __dsPIC30F6014__。

编译器将定义常量 __C30_VERSION__，给版本标识符赋一个数值。这可用于利用编译器新版本中与老版本向后兼容的特性。

这个值基于当前版本的主版本号和次版本号。例如，版本 2.00 将 __C30_VERSION__ 定义为 200。可以将这个宏与标准预处理比较语句一起使用，以便根据条件包含 / 排除各种代码结构。

可通过向命令行添加 --version 或查看版本附带的 README.TXT 文件，来确定 __C30_VERSION__ 的当前定义。

3.7.2 不赞成使用的常量

可在**附录 D “不赞成使用的特性”**中查到视为不赞成使用的常量。

3.8 通过命令行编译单个文件

本节说明如何编译和链接单个文件。为便于讨论，假定编译器安装在 c: 驱动器的 pic30-tools 目录中。因此就有下面的目录：

表 3-17： 与编译器有关的目录

目录	内容
c:\Program Files\Microchip\MPLAB C30\include	包含 ANSI C 头文件的目录。编译器在该目录中存放标准 C 函数库的系统头文件。PIC30_C_INCLUDE_PATH 环境变量指向这个目录。（在 DOS 命令提示符下键入 set 来检查这一点。）
c:\Program Files\Microchip\MPLAB C30\support\h	包含针对 dsPIC® DSC 器件的头文件的目录。编译器在该目录中存放针对 dsPIC DSC 器件的头文件。PIC30_C_INCLUDE_PATH 环境变量指向这个目录。（在 DOS 命令提示符下键入 set 来检查这一点。）
c:Program Files\Microchip\MPLAB C30\lib	库文件目录：该目录中存放库文件和预编译目标文件。
c:Program Files\Microchip\MPLAB C30\support\gld	链接描述文件目录：在这个目录中存放不同型号器件的链接描述文件。
c:Program Files\Microchip\MPLAB C30\bin	可执行文件目录：存放编译程序。 PATH 环境变量包含这个目录。

下面是一个两数相加的简单 C 程序。

使用任何文本编辑器创建下面的程序并保存为 ex1.c。

```
#include <p30f2010.h>
int main(void);
unsigned int Add(unsigned int a, unsigned int b);
unsigned int x, y, z;
int
main(void)
{
  x = 2;
  y = 5;
  z = Add(x,y);
  return 0;
}
unsigned int
Add(unsigned int a, unsigned int b)
{
  return(a+b);
}
```

程序的第一行包含了头文件 p30f2010.h，这个头文件提供了该器件的所有特殊功能寄存器的定义。关于头文件的详细信息，参见**第 6 章 "器件支持文件"**。

在 DOS 提示符下输入如下命令行来编译该程序：

```
C:\> pic30-gcc -o ex1.o ex1.c
```

命令行选项 -o ex1.cof 命名输出 COFF 可执行文件 （若未指定 -o 选项，则输出文件名为 a.exe）。 COFF 可执行文件可装载到 MPLAB IDE 中。

如果需要 hex 文件，如要装入器件编程器中，可以使用下面的命令：

```
C:\> pic30-bin2hex ex1.o
```

这样就生成了一个名为 ex1.hex 的 Intel hex 文件。

3.9 通过命令行编译多个文件

将 Add() 函数移到名为 add.c 的文件中来说明在一个应用程序中多个文件的使用。即：

文件 1

```
/* ex1.c */
#include <p30f2010.h>
int main(void);
unsigned int Add(unsigned int a, unsigned int b);
unsigned int x, y, z;
int main(void)
{
  x = 2;
  y = 5;
  z = Add(x,y);
  return 0;
}
```

文件 2

```
/* add.c */
#include <p30f2010.h>
unsigned int
Add(unsigned int a, unsigned int b)
{
  return(a+b);
}
```

在 DOS 提示符下输入如下命令行来编译这两个文件：

```
C:\> pic30-gcc -o ex1.o ex1.c add.c
```

这个命令编译模块 ex1.c 和 add.c。编译的模块和编译器库文件链接，并生成可执行文件 ex1.o。



第 4 章 MPLAB C30 C 编译器运行时环境

4.1 简介

本章讲述 MPLAB C30 C 编译器的运行时环境。

4.2 主要内容

本章讨论的内容包括：

- 地址空间
- 代码段和数据段
- 启动和初始化
- 存储空间
- 存储模型
- 定位代码和数据
- 软件堆栈
- C 堆栈使用
- C 堆使用
- 函数调用约定
- 寄存器约定
- 位反转寻址和模寻址
- 程序空间可视性 （PSV）的使用

4.3 地址空间

dsPIC 数字信号控制器 （DSC）器件融合了传统 PICmicro 单片机 （MCU）的特征 （外设、哈佛架构和 RISC）以及新的 DSP 功能。 dsPIC DSC 器件具有两个独立的存储器：

- 程序存储器 （图 4-1）包含可执行代码和常量数据。
- 数据存储器 （图 4-2）包含外部变量、静态变量、系统堆栈和数据寄存器。数据存储器由 near 数据和 far 数据组成，其中， near 数据指数据存储空间的前 8KB， far 数据指数据存储空间的上面 56KB。

尽管程序存储区和数据存储区是完全独立的，但编译器可通过程序空间可视性 （PSV）窗口访问程序存储器中的常量数据。

图 4-1： 程序存储空间映射

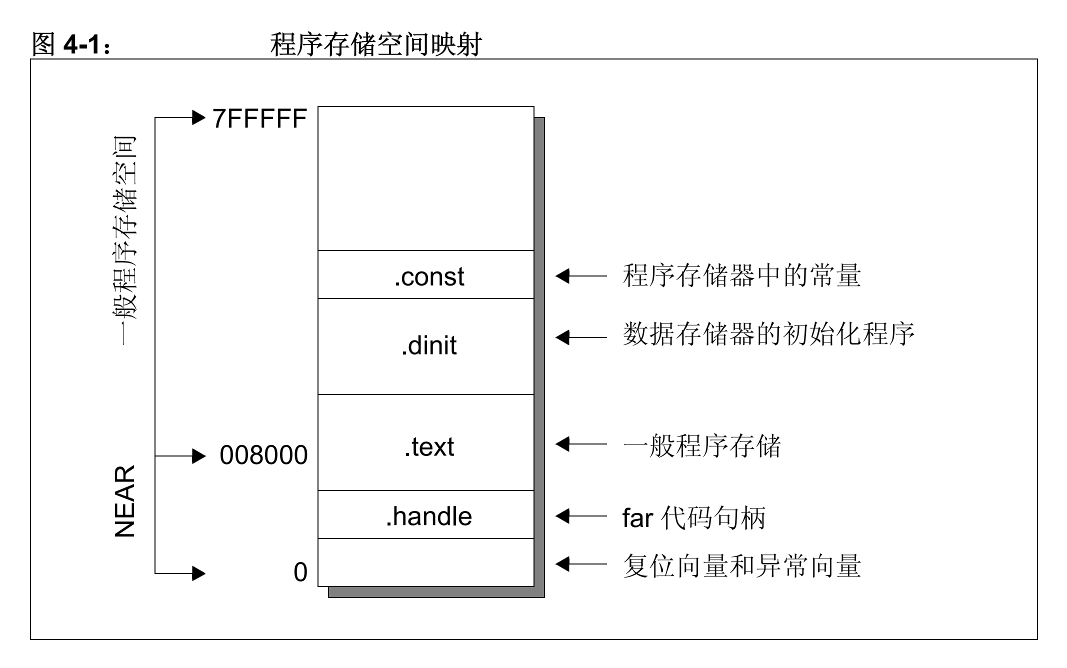
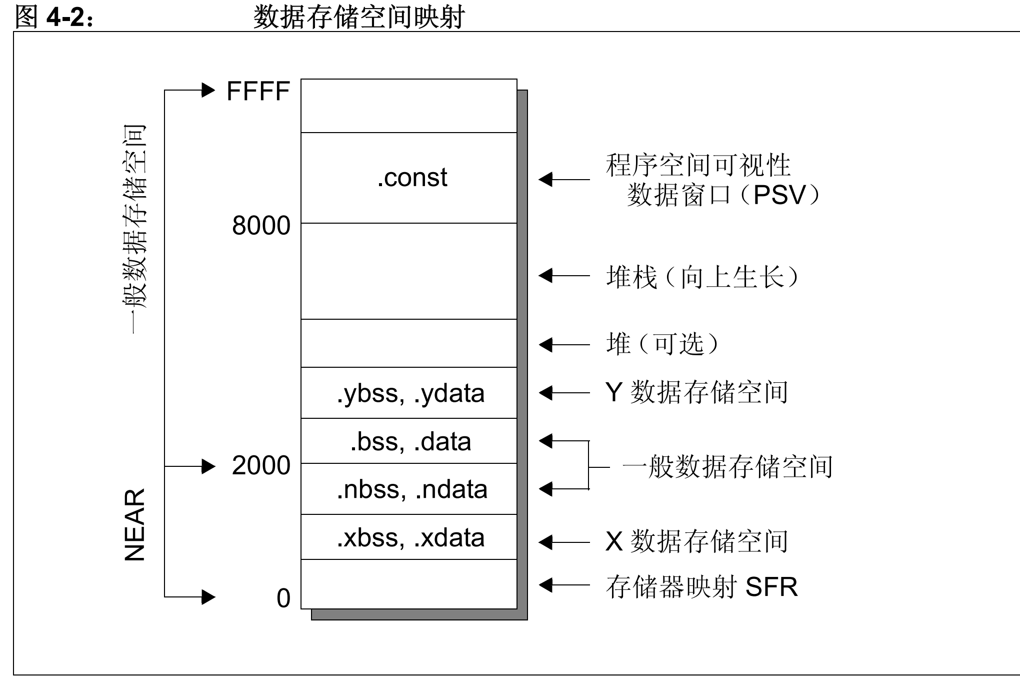


图 4-2： 数据存储空间映射



4.4 代码段和数据段

段是指占用 dsPIC DSC 器件存储器中连续地址的可定位代码或数据块。在任何给定的目标文件中，通常都有几个段。例如，一个文件中可能包含一个程序代码段、一个未初始化数据段及其他段。

除非通过段属性 （关于段属性的信息，参见**第 2.3 节 "关键字差别"**）指定，否则，MPLAB C30 编译器将代码和数据存放在默认的段中。所有由编译器生成的可执行代码都被分配到名为 `.text` 的段中，而数据则根据数据类型分配在不同的段中，见表 4-1。

表 4-1： 编译器生成的数据段

	已初始化		未初始化	
	变量	**ROM 中的常量**	**RAM 中的常量**	变量
near	.ndata	.const	.ndconst	.nbss
far	.data	.const	.dconst	.bss

下面列出了每个默认的段，并描述了存储到段中的信息的类型。

.text

可执行代码分配到 `.text` 段中。

.data

具有 `far` 属性的已初始化变量分配到 `.data` 段中。当选择大数据存储模型时 （即使用 `-mlarge-data` 命令行选项时），这是已初始化变量的默认位置。

.ndata

具有 `near` 属性的已初始化变量分配到 `.ndata` 段中。当选择小数据存储模型时 （即使用默认的 `-msmall-data` 命令行选项时），这是已初始化变量的默认段。

.const

当使用默认的 `-mconst-in-code` 命令行选项时，常量值，如字符串常量和 `const` 限定的变量，分配到 `.const` 段中。这种段位于程序存储器中并通过 PSV 窗口访问。

还可以通过段属性，不需要在命令行中使用 `-mconst-in-code` 选项，将变量存放到 `.const` 段中：

```
int i __attribute__((space(auto_psv)));
```

.dconst

当使用 -mlarge-data 命令行选项时，不需使用 -mconst-in-code 选项，即可将常量值，如字符串常量和 const 限定的变量分配到 .dconst 段中。 MPLAB C30 启动代码将通过从 .dinit 段复制数据来初始化这种段，除非指定了链接器选项 --no-data-init。 .dinit 段由链接器生成，分配到程序存储器中。

.ndconst

当使用默认的 -msmall-data 命令行选项时，不需使用 -mconst-in-code 命令行选项，即可将常量值，如字符串常量和 const 限定的变量分配到 .ndconst 段中。 MPLABC30 启动代码将通过从 .dinit 段中复制数据来初始化这种段。 .dinit 段由链接器生成，位于程序存储器中。

.bss

具有 far 属性的未初始化变量分配到 .bss 段中。当选择了大数据存储模型时 （即当使用 -mlarge-data 命令行选项时），这是未初始化变量的默认位置。

.nbss

具有 near 属性的未初始化变量分配到 .nbss 段中。当选择小数据存储模型时 （即使用默认的 -msmall-data 命令行选项时），这是未初始化变量的默认位置。

.pbss——持久数据

如果应用需要将数据存储在 RAM 中而不受器件复位的影响，可以使用段 .pbss。段 .pbss 分配到 near 数据存储区中，不会被 libpic30.a 中的默认启动模块修改。

可使用段属性将未初始化变量存放在 .pbss 段中：

```
int i __attribute__((persistent));
```

为了利用持久数据存储， main() 函数的开头要检测所发生复位的类型。在 RCON 复位控制寄存器中的各位可用于检测复位源。详细信息请参阅 *《dsPIC30F 系列参考手册》*（DS70046D_CN）的第 8 章。

4.5 启动和初始化

libpic30.a 归档 / 库中包含两个 C 运行时启动模块。这两个启动模块的入口点都是 __reset。链接描述文件在程序存储器的地址 0 存放了一条 GOTO __reset 指令，在器件复位时转移控制。

默认情况下链接主启动模块并进行以下操作：

1. 使用链接器或用户定义链接描述文件提供的值对堆栈指针 （W15）和堆栈指针限制寄存器 （SPLIM）进行初始化。详细信息参见**第 4.9 节 "软件堆栈"**。
2. 如果定义了.const段，那么将通过初始化PSVPAG和CORCON寄存器将其映射到程序空间可视性窗口。注意，当在 MPLAB IDE 中选择了 "Constants in code space" 选项或在 MPLAB C30 命令行中指定了默认的 -mconst-in-code 选项时，将定义一个 .const 段。
3. 读取 .dinit 段中的数据初始化模板，会导致所有未初始化的段被清零，同时所有已初始化段被初始化为从程序存储器中读取的值。数据初始化模板由链接器创建，并支持**第 4.4 节 "代码段和数据段"**中列出的标准段和用户定义段。

注：	持久数据段 .pbss 不会被清零或初始化。

4. 调用 main 函数时不带参数。
5. 如果从 main 函数返回，处理器将复位。

当指定 -Wl、 --no-data-init 选项时，将链接备用启动模块 （crtl.o）。它执行和上面相同的操作，除了第 3 步，这一步省略掉。备用启动模块比主启动模块小，所以当不需要初始化数据时，可选择该模块来节省程序存储空间。

这两个模块的源代码 （采用 dsPIC DSC 汇编语言）存放在 c:\Program Files\Microchip\MPLAB C30\src 目录中。如果需要，可以对启动模块进行修改。例如，如果应用需要在调用 main 函数时带参数，可通过改变条件汇编伪指令来提供这一支持。

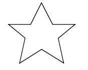
4.6 存储空间

静态和外部变量通常分配到一般数据存储区中。如果选择了 constants-in-data 存储模型， const 限定的变量将被分配到一般数据存储区中；如果选择了 constants-in-code 存储模型， const 限定的变量将被分配到程序存储器中。

为配合 dsPIC DSC 的架构特点， MPLAB C30 定义了几个专用存储空间。可通过使用 `space` 属性 （参见第 **2.3.1 节 "指定变量的属性"**），将静态和外部变量分配到专用存储空间中：

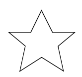
`data`

一般数据空间。可使用普通 C 语句访问一般数据空间中的变量。这是默认的分配。



`xmemory`——仅适用于 dsPIC30F/33F DSC

X 数据地址空间。可使用普通 C 语句访问 X 数据空间中的变量。 X 数据地址空间尤其适用于针对 DSP 的函数库和 / 或汇编语言指令。



`ymemory`——仅适用于 dsPIC30F/33F 器件

Y 数据地址空间。可使用普通 C 语句访问 Y 数据空间中的变量。Y 数据地址空间尤其适用于针对 DSP 的函数库和 / 或汇编语言指令。

`prog`

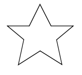
一般程序空间，通常保留给可执行代码。不能使用普通 C 语句访问程序空间中的变量。这些变量必须由编程人员显式访问，通常通过表访问行内汇编指令，或使用程序空间可视性窗口访问。

`const`

程序空间中编译器管理的区域，用于程序空间可视性窗口访问。 const 空间中的变量可通过普通 C 语句读取，总分配空间最大为 32K。

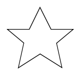
`psv`

程序空间，用于程序空间可视性窗口访问。 PSV 空间中的变量不由编译器管理，不能通过普通 C 语句访问。这些变量必须由编程人员显式访问，通常通过表访问行内汇编指令，或使用程序空间可视性窗口访问。可通过使用 PSVPAG 寄存器的设置，访问 PSV 空间中的变量。



`eedata`——仅适用于 dsPIC30F/33F 器件

数据 EEPROM 空间，位于程序存储器高地址的 16 位宽非易失性存储区。 eedata 空间中的变量不能使用普通 C 语句访问。这些变量必须由编程人员显式访问，通常使用表访问行内汇编指令，或使用程序空间可视性窗口访问。



`dma`——仅适用于 PIC24H MCU 和 dsPIC30F/33F DSC

DMA 存储器。可使用一般 C 语句或通过 DMA 外设访问 DMA 存储器中的变量。

4.7 存储模型

编译器支持几种存储模型。提供了命令行选项来根据所使用的特定 dsPIC DSC 器件和存储器类型，选择最佳的存储模型。

表 **4-2**： 存储模型命令行选项

选项	存储区定义	描述
`-msmall-data`	8 KB 数据存储区。 **这是默认设置。**	允许使用类 PIC18 指令访问数据存储器。
`-msmall-scalar`	8 KB 的数据存储区。 **这是默认设置。**	允许使用类 PIC18 指令访问数据存储器中的标量。
`-mlarge-data`	大于 8 KB 的数据存储区。	使用数据引用伪指令。
`-msmall-code`	32 K 字的程序存储区。 **这是默认设置。**	函数指针不使用跳转表。函数调用使用 `RCALL` 指令。
`-mlarge-code`	大于 32 K 字的程序存储区。	函数指针使用跳转表。函数调用使用 `CALL` 指令。
`-mconst-in-data`	位于数据存储器中的常量。	由启动代码从程序存储器中复制的值。
`-mconst-in-code`	位于程序存储器中的常量。 **这是默认设置。**	通过程序空间可视性 （PSV）数据窗口访问这些值。

命令行选项适用于所有被编译的模块。各个变量和函数可以声明为 `near` 或 `far`，以便更好的控制代码的生成。关于设置变量和函数属性的信息，请参阅**第 2.3.1 节 “指定变量的属性”**和**第 2.3.2 节 “指定函数的属性”**。

4.7.1 Near 数据和 Far 数据

如果变量分配到 near 数据段中，通常编译器能生成更好 （更紧凑）的代码。如果一个应用的所有变量能存放在 8KB 的 near 数据存储区中，那么编译器在编译每个模块时就会被要求使用默认的 `-msmall-data` 命令行选项来将这些变量存放在 near 数据存储区中。如果标量类型 （非数组或结构类型）所占用的数据总量小于 8KB 的话，可使用默认的 `-msmall-scalar` 选项。这要求编译器仅将应用中的标量存放在 near 数据段中。

如果这些全局选项都不适合，那么就使用下面的可选方案：

1. 可以使用 -mlarge-data 或 -mlarge-scalar 命令行选项编译应用的某些模块。在这种情况下，仅这些模块使用的变量被分配到 far 数据段中。如果使用这个可选方案，在使用外部定义的变量时一定要小心。如果使用这两个命令行选项之一编译的模块使用了一个外部定义的变量，则定义这个变量的模块也要使用相同的选项编译，或者在变量声明和定义时指定 far 属性。
2. 如果使用了 -mlarge-data 或 -mlarge-scalar 命令行选项，那么可通过指定 near 属性将变量排除在 far 数据空间外 （即存放在 near 数据空间）。
3. 命令行选项的作用域仅限于模块内部，也可以不使用命令行选项，而通过指定 far 属性将变量存放在 far 数据段中。

如果应用的 near 变量在 8K 的 near 数据空间中存放不下，链接器将产生错误消息。

4.7.2 Near 代码和 Far 代码

具有 near 属性的函数 （函数在彼此的 32K 字范围内）互相调用时比非 near 属性的函数效率高。如果已知应用程序中的所有函数都是 near 的，那么在编译每个模块时就可以使用默认的 -msmall-code 命令行选项来指示编译器采用更高效的函数调用形式。

如果这个默认的选项不适合，可以使用下面的可选方案：

1. 可以使用 -msmall-code 命令行选项来编译应用程序的某些模块。在这种情况下，只有这些模块中的函数调用可以采用更高效的函数调用形式。
2. 如果已使用-msmall-code命令行选项，那么编译器可能被指示对具有far属性的函数使用长函数调用形式。
3. 命令行选项的作用域限于模块内部，可以不使用命令行选项，而通过在函数的定义和声明中指定 near 属性，指示编译器使用更高效的函数调用形式调用这些函数。

-msmall-code 命令行选项与 -msmall-data 命令行选项的区别在于，采用前者时，为确保函数彼此 “靠近”分配，编译器不需进行特别的操作；而采用后者时，编译器要将变量分配到特殊的段中。

如果函数声明为 near，而其调用函数无法采用函数调用的更高效形式调用它时，链接器将产生错误信息。

4.8 定位代码和数据

正如**第 4.4 节 “代码段和数据段”**中所述，编译器将代码存放在 .text 段中，而根据所使用的存储模型和数据是否已初始化将数据存放在指定的段中。链接模块时，链接器根据各个段的属性来确定段的起始地址。

某些情况下必须将特定函数或变量存放在某个特定地址或某个地址范围。为实现这一点，最简单的方法是使用 address 属性，如**第 2.3 节 “关键字差别”**所述。例如，将函数存放到程序存储器的地址 0x8000 中：

```
int __attribute__ ((address(0x8000))) PrintString (const char *s);
```

同样，将变量 Mabonga 存放到数据存储器的地址 0x1000 中：

```
int __attribute__ ((address(0x1000))) Mabonga = 1;
```

定位代码和数据的另一种方法是将函数或变量存放到用户定义的段中，并在自定义的链接描述文件中指定该段的起始地址。具体如下：

1. 在 C 源程序中修改代码或数据的声明来指定用户定义的段。
2. 将这个用户定义段加入到一个自定义的链接描述文件中来指定段的起始地址。

例如，要将函数 PrintString 存放到程序存储器的 0x8000 地址中，首先要在 C 源程序中对函数进行如下声明：

```
int __attribute__((__section__(".myTextSection")))
PrintString(const char *s);
```

段属性指定将函数存放到名为 .myTextSection 的段中，而不是默认的 .text 段中。它没有指定用户定义的段存存放在哪里。这必须在一个自定义的链接描述文件中指定，如下所示。以针对器件的链接描述文件为基础，加入如下段定义：

```
.myTextSection 0x8000 :
  {
        *(.myTextSection);
  } >program
```

这指定了输出文件应包括一个名为 .myTextSection 的段，这个段位于地址 0x8000，包含所有名为 .myTextSection 的输入段。由于在本例中，在该段中只有一个函数 PrintString，那么这个函数将位于程序存储器的地址 0x8000 处。

类似地，要将变量 Mabonga 存放到数据存储器的地址 0x1000 中，首先要在 C 源程序中声明该变量如下：

```
int __attribute__((__section__(".myDataSection"))) Mabonga =
1;
```

段属性指定将变量存放到名为 .myDataSection 的段中，而不是默认的 .data 段中。它没有指定用户定义的段存放在哪里。同样地，这必须在一个自定义的链接描述文件中指定，如下所示。以特定器件的链接描述文件为基础，加入如下段定义：

```
.myDataSection 0x1000 :
  {
        *(.myDataSection);
  } >data
```

这指定了输出文件应包含一个名为 .myDataSection 的段，这个段位于地址 0x1000，包含所有名为 .myDataSection 的输入段。在这个例子中，由于该段中仅包含一个变量 Mabonga，那么该变量将被存放到数据存储器的地址 0x1000 中。

4.9 软件堆栈

dsPIC DSC 器件的寄存器 W15 专门用作软件堆栈指针。所有处理器堆栈操作，包括函数调用、中断和异常都使用软件堆栈。堆栈是向上生长的，向高存储地址生长。

dsPIC DSC 器件也支持堆栈溢出检测。如果堆栈指针限制寄存器 SPLIM 已被初始化，器件将对所有堆栈操作的溢出进行检测。如果发生溢出，处理器将启动一个堆栈错误异常处理。默认情况下，这将引起处理器复位。应用还可通过定义一个名为 _StackError 的中断函数来安装一个堆栈错误异常处理程序。详细信息请参阅**第 7 章 "中断"**。

C 运行时启动模块在启动和初始化过程中对堆栈指针 （W15）和堆栈指针限制寄存器（SPLIM）进行初始化。初值通常由链接器提供，链接器尽可能在未使用的数据存储器中分配最大的堆栈。链接映射输出文件中给出堆栈的地址。通过 -stack 链接器命令行选项，应用可确保至少可获得一个最小的堆栈。详细信息请参阅 *《MPLAB® ASM30、 MPLAB LINK30 和实用程序用户指南》*（DS51317D_CN）。

另外，可以通过自定义链接描述文件中的用户定义段来分配指定大小的堆栈。在下面的示例中， 0x100 字节的数据存储区保留给了堆栈。声明了用于 C 运行时启动模块的两个符号， __SP_init 和 __SPLIM_init。

```
.stack :
  {
      __SP_init = .;
      . += 0x100
      __SPLIM_init = .;
    . += 8
  } >data
```

__SP_init 定义了堆栈指针 （W15）的初值，而 __SPLIM_init 定义了堆栈指针限制寄存器 （SPLIM）的初值。 __SPLIM_init 的值应比物理堆栈限制小至少 8 个字节，以便允许堆栈错误异常处理。如果安装了堆栈错误中断处理程序，由于要考虑到中断处理程序本身的堆栈使用， __SPLIM_init 的值应该更小。默认的中断处理程序不需要额外使用堆栈。

4.10 C 堆栈使用

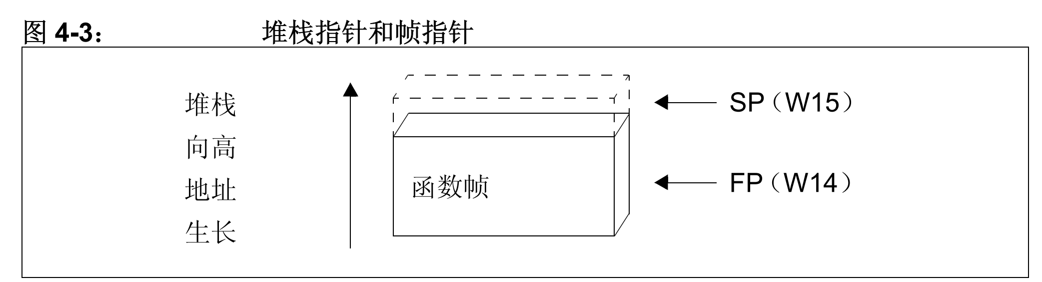
C 编译器使用软件堆栈来进行如下操作：

- 分配自动变量
- 传递函数参数
- 在中断函数中保存处理器状态
- 保存函数返回地址
- 存储临时变量
- 函数调用时保护寄存器

运行时堆栈是向上生长的，从低地址向高地址生长。编译器使用两个工作寄存器来管理堆栈：

- W15——这是堆栈指针。它指向栈顶，栈顶定义为堆栈的第一个未使用单元。
- W14——这是帧指针 （frame pointer， FP）。它指向当前函数的帧。如果需要，每个函数都会在栈顶创建一个新的帧来分配自动变量和临时变量。可使用编译器选项 `-fomit-frame-pointer` 来限制帧指针的使用。

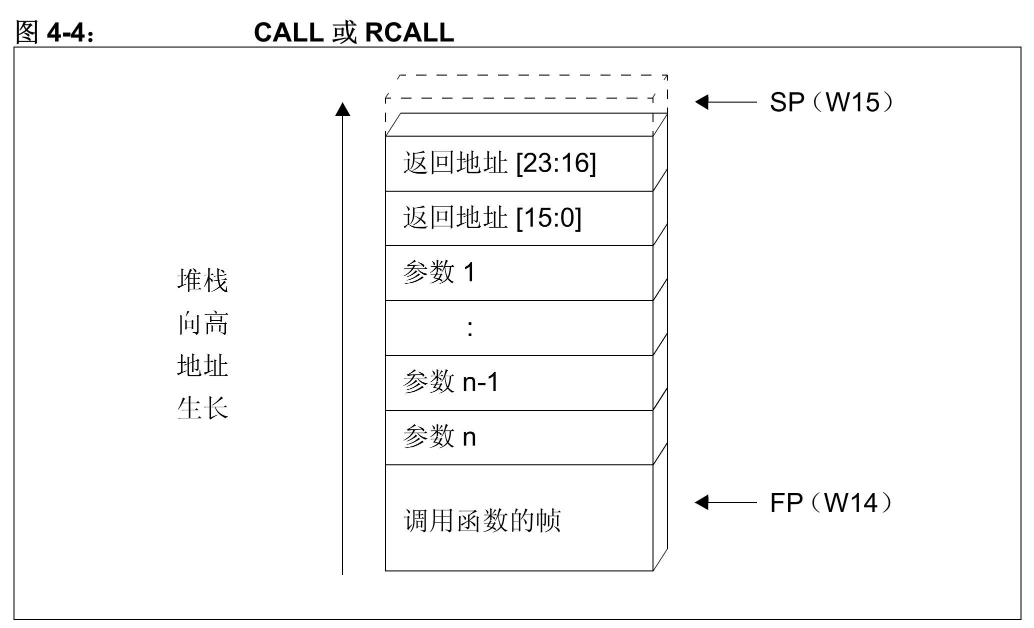
图 4-3： 堆栈指针和帧指针



C 运行时启动模块 （`libpic30.a` 中的 `crt0.o` 和 `crt1.o`）初始化堆栈指针 W15 使其指向栈底，初始化堆栈指针限制寄存器使其指向栈顶。堆栈向上生长，如果堆栈超出堆栈指针限制寄存器中的值，将转入堆栈错误陷阱。用户可以通过初始化堆栈指针限制寄存器来进一步限制堆栈的生长。

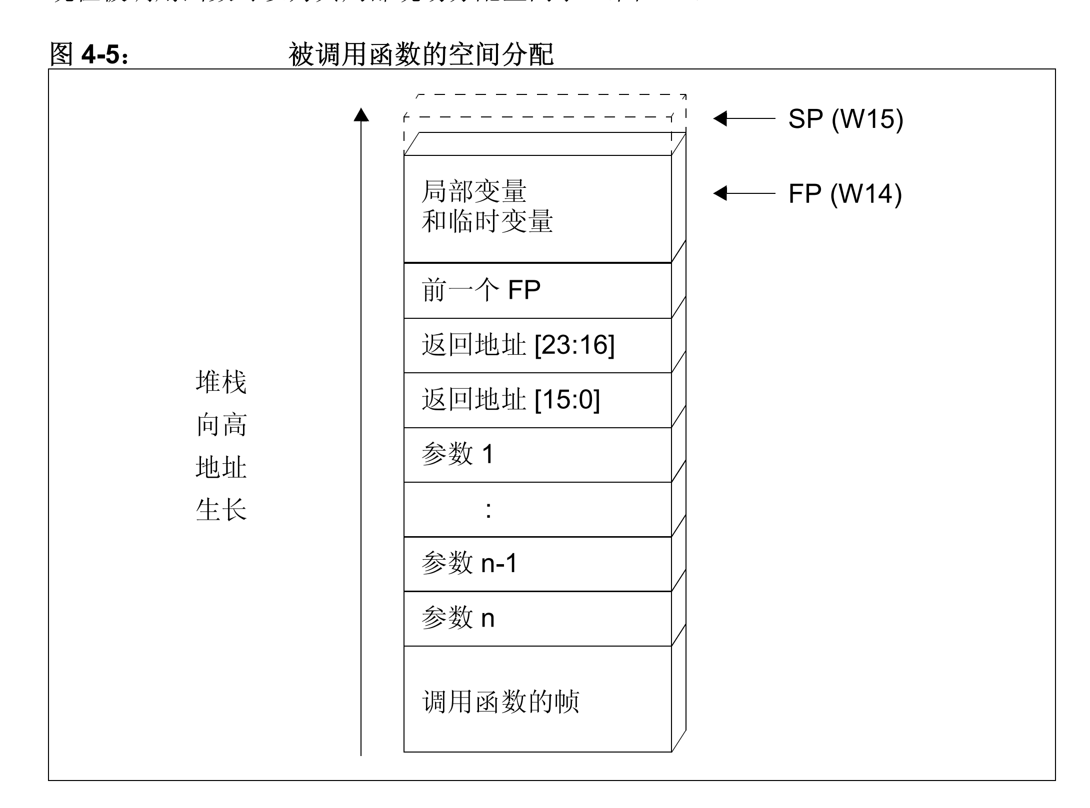
下图说明了调用一个函数的步骤。执行 `CALL` 或 `RCALL` 指令将返回地址压入软件堆栈。参见图 4-4。

图 4-4： CALL 或 RCALL



现在被调用函数可以为其局部现场分配空间了 （图 4-5）。

图 4-5： 被调用函数的空间分配



最后，函数中用到的所有被调用函数保存寄存器被压入堆栈 （图 4-6）。

图 4-6： **将被调用函数保存的寄存器压入堆栈**

4.11 C 堆使用

C 运行时堆是数据存储器中的未初始化区域，用于使用标准 C 函数库中的动态存储器管理函数 `calloc`、 `malloc` 和 `realloc` 进行动态存储器分配。如果不使用这些函数，就不需要分配堆。默认情况下不创建堆。

如果确实需要通过调用一个存储器分配函数直接使用动态存储器分配，或通过使用标准 C 函数库的输入 / 输出函数来间接使用动态存储器分配的话，就必须创建一个堆。通过使用链接器命令行选项 `--heap` 在链接器命令行中指定堆的大小即可创建一个堆。例如，使用命令行来分配一个 512K 字节的堆：

```
pic30-gcc foo.c -Wl,--heap=512
```

链接器会在堆栈下面分配一个堆 （图 4-2）。

如果使用标准 C 函数库的输入 / 输出函数，就必须分配一个堆。如果 `stdout` 是使用的唯一文件，那么堆的大小为 0，即使用命令行选项：

```
-Wl,--heap=0
```

如果打开几个文件，那么对于同时打开的每个文件，堆的大小必须包含 40 个字节。如果堆存储区的空间不足， `open` 函数将返回错误指示。对于每个应被缓存的文件，都需要 514 字节的堆空间。如果没有足够的堆存储区空间用于缓存，文件将以非缓存模式打开。

4.12 函数调用约定

调用函数时：

- 寄存器 W0-W7 由调用函数保存。为保存寄存器的值，调用函数必须将这些值压入堆栈。
- 寄存器 W8-W14 由被调用函数保存。被调用函数必须保存它会修改的任何这些寄存器。
- 寄存器 W0-W4 用于存放函数返回值。

表 4-3： 需要的寄存器

数据类型	需要的寄存器数
char	1
int	1
short	1
pointer	1
long	2 （邻近的 – 对齐到偶数编号的寄存器）
float	2 （邻近的 – 对齐到偶数编号的寄存器）
double*	2 （邻近的 – 对齐到偶数编号的寄存器）
long double	2 （邻近的 – 对齐到以 4 的倍数编号的寄存器）
structure	结构中 2 个字节使用 1 个寄存器

* 如果使用 -fno-short-double， double 等价于 long double。

参数存放到可用的第一批 （个）对齐的邻近寄存器中。如果需要的话，调用函数必须保存参数。结构没有任何对齐限制；如果有足够的寄存器来保存整个结构，结构参数将占用寄存器。函数结果存储在从 W0 开始的连续寄存器中。

4.12.1 函数参数

前八个工作寄存器 （W0-W7）用于存储函数参数。参数以自左向右的顺序分配到寄存器中，且参数被分配到对齐适当的第一个可用寄存器中。

下面的示例中，所有参数都通过寄存器传递，尽管这些参数不是以在声明中出现的顺序存放在寄存器中。这种格式允许 MPLAB C30 编译器最高效地使用可用的参数寄存器。

例 4-1： 函数调用模型

```
void
params0(short p0, long p1, int p2, char p3, float p4, void *p5)
{
        /*
        ** W0          p0
        ** W1          p2
        ** W3:W2       p1
        ** W4          p3
        ** W5          p5
        ** W7:W6       p4
        */
        ...
}
```


下面这个示例说明如何传递结构给函数。如果整个结构都可以存放在可用的寄存器中，那么通过寄存器来传递结构；否则结构参数将存放在堆栈中。

例 4-2：　　函数调用模型，传递结构

```
typedef struct bar {
  int i;
  double d;
} bar;

void
params1(int i, bar b) {
        /*
        ** W0          i
        ** W1          b.i
        ** W5:W2       b.d
        */

}
```

与长度可变参数列表中的省略号 （...）对应的参数不分配到寄存器中。任何不分配到寄存器的参数都以自右向左的顺序压入堆栈。

下面的示例中，由于结构参数太大而不能存放到寄存器中。但是，这并不会禁止使用寄存器来存放下一个参数。

例 4-3：　　函数调用模型，基于堆栈的参数

```
typedef struct bar {
  double d,e;
} bar;

void
params2(int i, bar b, int j) {
        /*
        ** W0           i
        ** stack        b
        ** W1           j
        */
}
```

对存放到堆栈的参数的访问与是否创建了帧指针有关。编译器一般情况下都创建帧指针 （除非已另外指示编译器不要这样做），将通过帧指针寄存器 （W14）访问基于堆栈的参数。在上面的示例中，通过 W14-22 访问 b。已通过减去上一个 FP 的 2 个字节，返回地址的 4 个字节，和 b 的 16 个字节，计算出相对于帧指针的偏移量为 -22（参见图 4-6）。

不使用帧指针时，汇编编程人员必须知晓自过程的入口开始使用了多少堆栈空间。如果没有额外使用堆栈空间，计算与上面类似。将通过 W15-20 访问 b：4 个字节用于返回地址， 16 个字节用于访问 b 的首地址。

4.12.2 返回值

8 位或 16 位标量的函数返回值返回到 W0 中， 32 位标量的函数返回值返回到 W1:W0 中，而 64 位标量的函数返回值返回到 W3:W2:W1:W0 中。聚集通过 W0 间接返回，W0 由调用函数设置为包含聚集值的地址。

4.12.3 调用函数时保存寄存器

对于一般的函数调用，编译器指定函数调用时保护寄存器 W8-W15。寄存器 W0-W7 可用作暂存寄存器。对于中断函数，编译器指定保护所有必需的寄存器，即 W0-W15 和 RCOUNT。

4.13 寄存器约定

特定寄存器在 C 运行时环境中起着特殊的作用。寄存器变量使用一个或多个工作寄存器，参见表 4-4。

表 **4-4:** 寄存器约定

变量	工作寄存器
`char, signed char, unsigned char`	W0-W13 和 W14 （如果没有用作帧指针的话）。
`short, signed short, unsigned short`	W0-W13 和 W14 （如果没有用作帧指针的话）。
`int, signed int,unsigned int`	W0-W13 和 W14 （如果没有用作帧指针的话）。
`void * (or any pointer)`	W0-W13 和 W14 （如果没有用作帧指针的话）。
`long, signed long, unsigned long`	一对邻近的寄存器，第一个寄存器是 {W0, W2, W4, W6, W8, W10, W12} 之一。低编号的寄存器包含值的最低 16 位。
`long long, signed long long, unsigned long long`	四个邻近的寄存器，第一个寄存器是 {W0, W4, W8} 之一。低编号的寄存器包含值的最低 16 位。接着的较高编号寄存器包含接着的较高位。
`float`	一对邻近的寄存器，第一个寄存器是 {W0, W2, W4, W6, W8, W10, W12} 之一。低编号的寄存器包含值的最低 16 位。
`double`*	四个邻近的寄存器，第一个寄存器是 {W0, W2, W4, W6, W8, W10, W12} 之一。低编号的寄存器包含值的最低 16 位。
`long double`	四个邻近的寄存器，第一个寄存器是 {W0, W4, W8} 之一。低编号的寄存器包含值的最低 16 位。

* 如果使用了 `-fno-short-double`， `double` 等价于 `long double`。

4.14 位反转寻址和模寻址

编译器并不直接支持位反转寻址和模寻址的使用。如果对一个寄存器使能了这两种寻址模式之一，那么编程人员要确保编译器不将该寄存器用作指针。当使能这两种寻址模式之一时，在产生中断时要特别小心。

可以在 C 中定义将在存储器中对齐适当的数组用于通过汇编语言函数进行模寻址。`aligned` 属性可用于定义用作递增模缓冲区的数组。`reverse` 属性可用于定义用作递减模缓冲区的数组。关于这些属性的详细信息，请参阅**第 2.3 节 "关键字差别"**。关于模寻址的更多信息，请参阅 *《dsPIC30F 系列参考手册》*（DS70046D_CN）的第 3 章。

4.15 程序空间可视性 （PSV）的使用

默认情况下，编译器自动将字符串和 `const` 限定的未初始化变量分配到映射到 PSV 窗口的 `.const` 段。然后 PSV 管理由编译器进行管理，编译器不会移动 PSV，将可访问程序存储区的大小限制为 PSV 窗口本身的大小。

或者，应用程序可控制 PSV 窗口来完成自己的功能。在应用程序直接控制 PSV 使用的优点是，这比将一个 `.const` 段永久映射到 PSV 窗口更为灵活。缺点是，应用程序必须管理 PSV 控制寄存器和位。指定 `-mconst-in-data` 选项来指示编译器不要使用 PSV 窗口。

`space` 属性可用来定义在 PSV 窗口中使用的变量 。为指定某些变量分配到编译器管理的段 `.const` 中，使用属性 `space(auto_psv)`。为将用于 PSV 访问的变量分配到不由编译器管理的段中，使用属性 `space(psv)`。关于这些属性的更多信息，请参阅**第 2.3 节 "关键字差别"**。

关于 PSV 使用的更多信息，请参阅 *《MPLAB® ASM30*、 *MPLAB LINK30 和实用程序用户指南》*（DS51317D_CN）。

注：

第 5 章　数据类型

5.1　简介

本章讲述 MPLAB C30 的数据类型。

5.2　主要内容

本章讲述的内容包括：

- 数据表示
- 整型
- 浮点型
- 指针

5.3　数据表示

多字节量以 " little endian"格式存储，即：

- 低字节存储在低地址中
- 低位存储在编号低的位地址中

例如， `0x12345678` 在地址 `0x100` 中存储如下：

0x100	0x78	0x56	0x101
0x102	0x34	0x12	0x103

而 `0x12345678` 在寄存器 W4 和 W5 中存储如下：

W4	W5
0x5678	0x1234

5.4　整型

表 5-1 列出了 MPLAB C30 所支持的整型数据类型。

表 5-1：　整型数据类型

类型	位	最小值	最大值
`char, signed char`	8	-128	127
`unsigned char`	8	0	255
`short, signed short`	16	-32768	32767
`unsigned short`	16	0	65535
`int, signed int`	16	-32768	32767
`unsigned int`	16	0	65535
`long, signed long`	32	-2^{31}	$2^{31} - 1$
`unsigned long`	32	0	$2^{32} - 1$
`long long**, signed long long**`	64	-2^{63}	$2^{63} - 1$
`unsigned long long**`	64	0	$2^{64} - 1$

** ANSI-89 扩展

关于实现定义的整型的信息，请参阅**第 A.6 节 "整型"**。

5.5 浮点型

MPLAB C30 使用 IEEE-754 格式。表 5-2 列出了所支持的浮点型数据类型。

表 5-2： 浮点型数据类型

类型	位	E 最小值	E 最大值	N 最小值	N 最大值
`float`	32	-126	127	2^{-126}	2^{128}
`double`*	32	-126	127	2^{-126}	2^{128}
`long double`	64	-1022	1023	2^{-1022}	2^{1024}

E = 指数
N = 归一化的 （近似值）
* 如果使用 `-fno-short-double`， `double` 等价于 `long double`。

关于实现定义的浮点型数的更多信息，请参阅**第 A.7 节 "浮点型"**。

5.6 指针

所有 MPLAB C30 指针都是 16 位宽的。这对于整个数据空间 （64 KB）的访问和小代码模型 （32 K 字的代码）足够了。在大代码模型 （大于 32 K 字的代码）中，指针可解析为 "句柄"；即，指针是位于程序空间前 32 K 字的 `GOTO` 指令的地址。



第 6 章 器件支持文件

6.1 简介

本章讲述为支持 MPLAB C30 编译使用的器件支持文件。

6.2 主要内容

本章讨论的内容包括：

- 处理器头文件
- 寄存器定义文件
- 使用特殊功能寄存器
- 使用宏
- 从 C 代码访问 EEDATA——仅适用于 dsPIC30F DSC

6.3 处理器头文件

处理器头文件随语言工具提供。这些头文件定义了每个 dsPIC DSC 器件中可用的特殊功能寄存器 （SFR）。要在 C 中使用头文件，使用：

```
#include <p30fxxxx.h>
```

其中 xxxx 对应器件的型号。 C 头文件包含在 support\h 目录中。

为使用特殊功能寄存器名 （如 CORCONbits），必须包含头文件。

例如下面的模块，是为 PIC30F2010 器件编写的，包括两个函数：一个函数用于使能 PSV 窗口，另一个函数用于禁止 PSV 窗口。

```
#include <p30f2010.h>
void
EnablePSV(void)
{
    CORCONbits.PSV = 1;
}
void
DisablePSV(void)
{
    CORCONbits.PSV = 0;
}
```

处理器头文件的约定是，使用器件数据手册中的寄存器名对每一个 SFR 命名。例如，CORCON 指内核控制寄存器。如果 SFR 中有一些重要的位，因此头文件中还有为该 SFR 定义的结构，结构名与 SFR 的名字相同，只是在后面附加了 “bits”。例如，CORCONbits 是内核控制寄存器的结构。使用数据手册中位的名字命名结构中的各位（或位域），如 PSV 表示 CORCON 寄存器中的 PSV 位。下面是对 CORCON 的完整定义 （将来可能会有所变化）：

```
/* CORCON: CPU Mode control Register */
extern volatile unsigned int CORCON __attribute__((__near__));
typedef struct tagCORCONBITS {
   unsigned IF      :1;    /* Integer/Fractional mode              */
   unsigned RND     :1;    /* Rounding mode                        */
   unsigned PSV     :1;    /* Program Space Visibility enable      */
   unsigned IPL3    :1;
   unsigned ACCSAT  :1;    /* Acc saturation mode                  */
   unsigned SATDW   :1;    /* Data space write saturation enable   */
   unsigned SATB    :1;    /* Acc B saturation enable              */
   unsigned SATA    :1;    /* Acc A saturation enable              */
   unsigned DL      :3;    /* DO loop nesting level status         */
   unsigned         :4;
} CORCONBITS;
extern volatile CORCONBITS CORCONbits __attribute__((__near__));
```

注：	符号 CORCON 和 CORCONbits 指同一个寄存器，在链接时将解析为同一地址。

6.4 寄存器定义文件

第 6.3 节 “处理器头文件”中描述的处理器头文件指定了每个器件的所有 SFR，但并未定义 SFR 的地址。 support\gld 目录中有每个器件的链接描述文件。链接描述文件定义了 SFR 的地址。要使用链接描述文件，指定链接器命令行选项：

```
-T p30fxxxx.gld
```

其中 xxxx 指器件的型号。

例如，假定有一个名为 app2010.c 的文件，它包含 PIC30F2010 器件的一个应用程序，则可使用下面的命令行编译和链接这个文件：

```
pic30-gcc -o app2010.o -T p30f2010.gld app2010.c
```

-o 命令行选项命名输出 COFF 可执行文件， -T 选项给出 PIC30F2010 器件的名称。如果在当前的目录中找不到 p30f2010.gld，链接器将在已知的库路径中搜索。对于默认的安装，链接描述文件包含在 PIC30_LIBRARAY_PATH 中。更多信息参见**第 3.6 节 “环境变量”**。

6.5 使用特殊功能寄存器

在应用程序中使用特殊功能寄存器时要遵循以下三个步骤：

1. 包含所使用器件的 处理器头文件。这样能提供该型号器件特殊功能寄存器的源代码。例如，下面的语句包含了 PIC30F6014 器件的头文件：

   ```
   #include <p30f6014.h>
   ```

2. 像访问任何其他 C 变量一样访问特殊功能寄存器。源代码可对特殊功能寄存器进行读和写。

   例如，下面的语句将 Timer1 特殊功能寄存器中的所有位清零。

   ```
   TMR1 = 0;
   ```

   下面一条语句中的 `T1CONbits.TON` 表示 T1CON 寄存器中的第 15 位，即 “定时器开启”位。这条语句将名为 `TON` 的位置 1 来启动定时器。

   ```
   T1CONbits.TON = 1;
   ```

3. 链接相应器件的寄存器定义文件或链接描述文件。链接器提供特殊功能寄存器的地址 （请记住在链接时位结构具有和 SFR 相同的地址）。例 6.1 将使用：

   ```
   p30f6014.gld
   ```

关于使用链接描述文件的更多信息，请参阅 *《MPLAB® ASM30、 MPLAB LINK30 和实用程序用户指南》*（DS51317D_CN）。

下面的例子是实时时钟的示例代码。它使用了几个特殊功能寄存器。这些特殊功能寄存器的描述参见 `p30f6014.h` 文件。该文件将和特定于器件的链接描述文件 `p30f6014.gld` 相链接。

例 6-1： 实时时钟示例代码

```
/*
** Sample Real Time Clock for dsPIC
**
**  Uses Timer1, TCY clock timer mode
**  and interrupt on period match
*/

#include <p30f6014.h>

/* Timer1 period for 1 ms with FOSC = 20 MHz */
#define TMR1_PERIOD 0x1388

struct clockType
  {
   unsigned int timer;     /* countdown timer, milliseconds */
   unsigned int ticks;     /* absolute time, milliseconds   */
   unsigned int seconds;   /* absolute time, seconds        */
  } volatile RTclock;

void reset_clock(void)
  {
   RTclock.timer = 0;      /* clear software registers  */
   RTclock.ticks = 0;
   RTclock.seconds = 0;

   TMR1 = 0;               /* clear timer1 register     */
   PR1 = TMR1_PERIOD;      /* set period1 register      */
   T1CONbits.TCS = 0;      /* set internal clock source */
   IPC0bits.T1IP = 4;      /* set priority level        */
   IFS0bits.T1IF = 0;      /* clear interrupt flag      */
   IEC0bits.T1IE = 1;      /* enable interrupts         */

   SRbits.IPL = 3;         /* enable CPU priority levels 4-7*/
   T1CONbits.TON = 1;      /* start the timer*/
  }
void __attribute__((__interrupt__)) _T1Interrupt(void)
  { static int sticks=0;

    if (RTclock.timer > 0)   /* if countdown timer is active */
        RTclock.timer -= 1;  /*  decrement it                */
    RTclock.ticks++;         /* increment ticks counter      */
    if (sticks++ > 1000)
    {                           /* if time to rollover          */
    sticks = 0;              /*  clear seconds ticks     */
    RTclock.seconds++;       /*  and increment seconds   */
    }

    IFS0bits.T1IF = 0;       /* clear interrupt flag     */
    return;
  }
```


6.6 使用宏

处理器头文件除定义了特殊功能寄存器外，还为 dsPIC30F 系列数字信号控制器定义了有用的宏。

6.6.1 配置位设置宏

提供了可用来设置配置位的宏。例如，为使用宏设置 FOSC 位，可在 C 源代码开头前插入下面的代码：

```
_FOSC(CSW_FSCM_ON & EC_PLL16);
```

这将使能外部时钟，PLL 设置为 16x。同时使能时钟切换和时钟故障保护监测。

同样，设置 FBORPOR 位：

```
_FBORPOR(PBOR_ON & BORV_27 & PWRT_ON_64 & MCLR_DIS);
```

这将使能 2.7 V 的欠压复位，将上电延时定时器初始化为 64 ms，并将 MCLR 引脚配置为普通 I/O 口。

每个配置位的设置列表，参见处理器头文件。

6.6.2 行内汇编使用的宏

下面列出了用于在 C 中定义汇编代码的宏：

```
#define Nop()    {__asm__ volatile ("nop");}
#define ClrWdt() {__asm__ volatile ("clrwdt");}
#define Sleep()  {__asm__ volatile ("pwrsav #0");}
#define Idle()   {__asm__ volatile ("pwrsav #1");}
```

6.6.3 数据存储器分配宏

本节讲述可用于分配数据存储空间的宏。有两种类型：需要参数的宏和不需要参数的宏。

下面的宏需要一个参数 N 来指定对齐。 N 必须是 2 的次幂，最小值为 2。

```
#define _XBSS(N)   __attribute__((space(xmemory), aligned(N)))
#define _XDATA(N)  __attribute__((space(xmemory), aligned(N)))
#define _YBSS(N)   __attribute__((space(ymemory), aligned(N)))
#define _YDATA(N)  __attribute__((space(ymemory), aligned(N)))
#define _EEDATA(N) __attribute__((space(eedata),  aligned(N)))
```

例如，声明一个未初始化数组位于 X 存储区中，对齐到 32 字节地址：

```
int _XBSS(32) xbuf[16];
```

声明一个未初始化数组位于数据 EEPROM 中，没有特殊对齐方式：

```
int _EEDATA(2) table1[] = {0, 1, 1, 2, 3, 5, 8, 13, 21};
```

下面的宏不需要参数。这些宏可用于将变量分配到持久数据存储区或 near 数据存储区中。

```
#define _PERSISTENT __attribute__((persistent))
#define _NEAR       __attribute__((near))
```

例如，声明器件复位后能保留其值的两个变量：

```
int _PERSISTENT var1,var2;
```

6.6.4 中断服务程序声明宏

下面的宏可用于声明中断服务程序 （ISR）：

```
#define _ISR __attribute__((interrupt))
#define _ISRFAST __attribute__((interrupt, shadow))
```

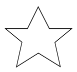
例如，声明 timer0 中断的中断服务程序：

```
void _ISR _INT0Interrupt(void);
```

声明 SPI1 中断的快速现场保护中断服务程序：

```
void _ISRFAST _SPI1Interrupt(void);
```

注：	如果中断服务程序使用了**第 7.4 节 “写中断向量”**中给出的保留名，中断服务程序的地址将自动填充到中断向量表中。



6.7 从 C 代码访问 EEDATA——仅适用于 dsPIC30F DSC

MPLAB C30提供了一些方便的宏定义来允许将数据存放在器件的EE数据区中。这实现起来很简单：

```
int _EEDATA(2) user_data[] =  { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

`user_data` 将被存放在 EE 数据空间中，为给定的初值保留 10 个字。

dsPIC DSC 器件为编程人员提供了两种方法来访问这种存储区。一种方法是通过程序空间可视性窗口。另一种方法是使用特殊机器指令 （TBLRDx）。

6.7.1 通过 PSV 访问 EEDATA

编译器通常管理 PSV 窗口来访问存储在程序存储器中的常量。否则， PSV 窗口可用于访问 EEDATA 存储器。

要使用 PSV 窗口：

- PSVPAG 寄存器必须设置为要访问的程序存储器的正确地址。对于 EE 数据，该地址为 0xFF，但最好使用 `__builtin_psvpage()` 函数。
- 还应通过置位 CORCON 寄存器中的 PSV 位来使能 PSV 窗口。如果该位没有置位，使用 PSV 窗口将始终读为 0x0000。

例 6-2： 通过 PSV 访问 EEDATA

```
#include <p30fxxxx.h>
int main(void) {
  PSVPAG = __builtin_psvpage(&user_data);
  CORCONbits.PSV = 1;

  /* ... */

  if (user_data[2]) ;/* do something */

  }
```


这些步骤仅需要执行一次。除非改变了 PSVPAG，否则可通过像普通 C 变量一样引用 EE 数据空间中的变量来读这些变量，如本例。

注：	这一访问模式与编译器管理的 PSV （-mconst-in-code）模式不兼容。要注意防止出现冲突。

6.7.2 使用 TBLRDx 指令访问 EEDATA

编译器不直接支持 TBLRDx 指令，但可以通过行内汇编使用这些指令。像 PSV 访问一样，通过一个 SFR 值形成一个 23 位的地址，并将地址编码为指令的一部分。为访问与前面示例中的存储器，可使用下面的代码：

要使用 TBLRDx 指令：

- TBLPAG 寄存器必须设置为要访问的程序存储器的正确地址。对于 EE 数据存储器，这一地址为 0x7F，但最好使用 `__builtin_tblpage()` 函数。
- TBLRDx 指令只能通过 `__asm__` 语句访问。关于这一指令的信息，请参阅 *《dsPIC30F/33F 程序员参考手册》*（DS70157B_CN）。

例 6-3： 通过表读访问 EEDATA

```
#include <p30fxxxx.h>

#define eedata_read(src, dest) {    \
  register int eedata_addr;         \
  register int eedata_val;          \
                                    \
  eedata_addr = __builtin_tbloffset(&src); \
  __asm__("tblrdl [%1], %0" : "=r"(eedata_val) : "r"(eedata_addr)); \
  dest = eedata_val;                \
  }

int main(void) {
  int value;

  TBLPAG = __builtin_tblpage(&user_data);

  eedata_read(user_data[2], value);
  if (value) ; /* do something */

  }
```

6.7.3 其他信息来源

《dsPIC30F 系列参考手册》（DS70046D_CN）的第 5 章对 dsPIC DSC 器件提供的闪存程序存储器和 EE 数据存储器的使用进行了很好的论述。本章也包含关于程序存储器和 EE 数据存储器运行时编程的信息。

注：



第 7 章 中断

7.1 简介

中断处理对于大多数单片机应用来说都是很重要的一个方面。中断用来使软件操作与实时发生的事件同步。当发生中断时，软件的正常执行流程被打断，调用专门的函数来处理事件。当中断处理结束时，恢复先前的现场信息并继续正常执行流程。

dsPIC30F 器件支持多个内部和外部中断源。另外，允许高优先级中断中断任何正在处理的低优先级中断。

MPLAB C30 编译器完全支持在 C 或行内汇编代码中进行中断处理。本章将对中断处理做一个概括介绍。

7.2 主要内容

本章讨论的主题包括：

- **编写中断服务程序** — 可以将一个或多个 C 函数指定为中断服务程序（ISR），在发生中断时调用。为了获得最好的性能，通常将长的计算或需要调用库函数的操作放在主应用程序中。当中断事件发生很快时，这种方式可以优化性能并最大程度降低信息丢失的可能性。
- **写中断向量** — 当发生中断时，dsPIC30F 器件使用中断向量来转移应用控制。中断向量是程序存储器中的专用地址，指定 ISR 的地址。为使用中断，应用必须在这些地址中包含有效的函数地址。
- **中断服务程序现场保护** — 为了保证从中断返回到代码后，条件状态与中断前相同，必须保护特定寄存器的现场信息。
- **中断响应时间** — 从中断事件发生到执行 ISR 第一条指令之间的时间就是中断响应时间。
- **中断嵌套** — MPLAB C30 支持中断嵌套。
- **允许 / 禁止中断** — 通过两种方式允许和禁止中断源：全局允许和单独允许。
- **中断服务程序和一般代码共用存储空间** — 采用这种方法时如何降低潜在的危险。

7.3 编写中断服务程序

可以根据本节提供的要领，仅使用 C 语言语法结构编写所有应用代码，其中包括中断服务程序。

7.3.1 编写中断服务程序的要领

编写 ISR 的要领为：

- 不带参数并以 void 返回值类型声明 ISR （强制）
- 不要通过一般程序调用 ISR （强制）
- 不要用 ISR 调用其他函数 （建议）

MPLAB C30 的 ISR 和任何其他 C 函数一样，可以有局部变量，可以访问全局变量。但是，ISR 需要声明为没有参数，没有返回值。这是必须的，因为 ISR 作为对硬件中断或陷阱的响应，对它的调用与一般 C 程序异步 （即 ISR 不是按通常的方式调用的，因此不能有参数和返回值）。

ISR 只能通过硬件中断或陷阱调用，不能通过其他 C 函数调用。 ISR 使用中断返回（RETFIE）指令退出函数，而不是使用一般的 RETURN 指令。不恢复现场使用 RETFIE 指令退出中断服务程序会破坏处理器资源，如 Status 寄存器的值。

最后，由于中断响应时间的原因，建议不要使用 ISR 调用其他函数。更多信息请参阅**第 7.6 节 “中断响应时间”**。

7.3.2 编写中断服务程序的语法

为将 C 函数声明为中断服务程序，指定函数的 interrupt 属性 （参见第 2.3 节中关于 __attribute__ 关键字的描述）。 interrupt 属性的语法如下：

```
__attribute__((interrupt [(
                            [ save(symbol-list)]
                            [, irq(irqid)]
                            [, altirq(altirqid)]
                            [, preprologue(asm)]
                            )]
              ))
```

可以在 interrupt 属性名和参数名的前后加下划线字符。因此， interrupt 和 __interrupt__ 是等价的， save 和 __save__ 也是等价的。

可选的 save 参数指定进入和退出 ISR 时需要保护和恢复的一个或多个变量。变量名列表包含在括号内，变量名中间用逗号分隔开。

如果不想导出全局变量的值，应该保护可能在 ISR 中修改的全局变量。被 ISR 修改的全局变量应该用 volatile 限定。

可选的 irq 参数允许将一个中断向量对应于一个特定的中断，可选的 altirq 参数允许将一个中断向量对应于一个指定的备用中断。每个参数都需要一个括号括起来的中断 ID 号。（参阅**第 7.4 节 “写中断向量”**中的中断 ID 列表。）

可选的 preprologue 参数允许在生成的代码中，编译器生成的函数 prologue 前插入汇编语句。

7.3.3 为中断服务程序编写代码

下面的原型声明了函数 isr0 为中断服务程序：

```
void __attribute__((__interrupt__)) isr0(void);
```

由原型可以看出，中断函数必须不带参数，没有返回值。如果需要的话，编译器将保护所有工作寄存器，以及 Status 寄存器和重复计数寄存器。课通过将其他变量指定为 interrupt 属性的参数保护这些变量。例如，要使编译器自动保护和恢复变量 var1 和 var2，使用下面的原型：

```
void __attribute__((__interrupt__(__save__(var1,var2))))
isr0(void);
```

为请求编译器使用快速现场保护 （使用 push.s 和 pop.s 指令），指定函数的 shadow 属性 （参阅**第 2.3.2 节 "指定函数的属性"**）。例如：

```
void __attribute__((__interrupt__, __shadow__)) isr0(void);
```

7.3.4 使用宏声明简单的中断服务程序

如果一个中断服务程序不需要 interrupt 属性的任何可选参数，则可使用简单的语法。在针对器件的头文件中定义了下面的宏：

```
#define _ISR __attribute__((interrupt))
#define _ISRFAST __attribute__((interrupt, shadow))
```

例如，声明 timer0 中断的中断服务程序：

```
#include <p30fxxxx.h>
void _ISR _INT0Interrupt(void);
```

用快速现场保护声明 SPI1 中断的中断服务程序：

```
#include <p30fxxxx.h>
void _ISRFAST _SPI1Interrupt(void);
```

7.4 写中断向量

dsPIC30F/33F DSC 和 PIC24F/H MCU 器件有两个中断向量表 — 主表和备用表 — 每个表都包含若干异常向量。

异常源有主异常向量和备用异常向量与之相关联，每个向量占据一个程序字，如下面这些表所示。当 INTCON2 寄存器中的 ALTIVT 位置位时，使用备用向量名。

> **注：** 器件复位不通过中断向量表处理。而是在器件复位时，清零程序计数器。这使处理器从地址 0 处开始执行。按照约定，链接描述文件在该地址处构建 GOTO 指令来转移控制到 C 运行时启动模块。

- 表 7-1 中断向量——dsPIC30F DSC （非 SMPS）
- 表 7-2 中断向量——dsPIC30F DSC （SMPS）
- 表 7-3 中断向量——PIC24F MCU
- 表 7-4 中断向量——dsPIC33F DSC/PIC24H MCU

表 7-1： 中断向量——dsPIC30F DSC （非 SMPS）

IRQ#	主向量名	备用向量名	向量函数
N/A	_ReservedTrap0	_AltReservedTrap0	保留
N/A	_OscillatorFail	_AltOscillatorFail	振荡器故障陷阱
N/A	_AddressError	_AltAddressError	地址错误陷阱
N/A	_StackError	_AltStackError	堆栈错误陷阱
N/A	_MathError	_AltMathError	数学错误陷阱
N/A	_ReservedTrap5	_AltReservedTrap5	保留
N/A	_ReservedTrap6	_AltReservedTrap6	保留
N/A	_ReservedTrap7	_AltReservedTrap7	保留
0	_INT0Interrupt	_AltINT0Interrupt	INT0 外部中断 0
1	_IC1Interrupt	_AltIC1Interrupt	IC1 输入捕捉 1
2	_OC1Interrupt	_AltOC1Interrupt	OC1 输出比较 1
3	_T1Interrupt	_AltT1Interrupt	TMR1 Timer 1 超时
4	_IC2Interrupt	_AltIC2Interrupt	IC2 输入捕捉 2
5	_OC2Interrupt	_AltOC2Interrupt	OC2 输出比较 2
6	_T2Interrupt	_AltT2Interrupt	TMR2 Timer 2 超时
7	_T3Interrupt	_AltT3Interrupt	TMR3 Timer 3 超时
8	_SPI1Interrupt	_AltSPI1Interrupt	SPI1 串行外设接口 1
9	_U1RXInterrupt	_AltU1RXInterrupt	UART1RX Uart 1 接收器
10	_U1TXInterrupt	_AltU1TXInterrupt	UART1TX Uart 1 发送器
11	_ADCInterrupt	_AltADCInterrupt	ADC 转换完成
12	_NVMInterrupt	_AltNVMInterrupt	NMM NVM 写完成
13	_SI2CInterrupt	_AltSI2CInterrupt	从 I²C™ 中断
14	_MI2CInterrupt	_AltMI2CInterrupt	主 I²C™ 中断
15	_CNInterrupt	_AltCNInterrupt	CN 输入变化中断
16	_INT1Interrupt	_AltINT1Interrupt	INT1 外部中断 1
17	_IC7Interrupt	_AltIC7Interrupt	IC7 输入捕捉 7
18	_IC8Interrupt	_AltIC8Interrupt	IC8 输入捕捉 8
19	_OC3Interrupt	_AltOC3Interrupt	OC3 输出比较 3

表 **7-1**： 中断向量——**dsPIC30F DSC** （非 **SMPS**）（续）

IRQ#	主向量名	备用向量名	向量函数
20	_OC4Interrupt	_AltOC4Interrupt	OC4 输出比较 4
21	_T4Interrupt	_AltT4Interrupt	TMR4 Timer 4 超时
22	_T5Interrupt	_AltT5Interrupt	TMR5 Timer 5 超时
23	_INT2Interrupt	_AltINT2Interrupt	INT2 外部中断 2
24	_U2RXInterrupt	_AltU2RXInterrupt	UART2RX Uart 2 接收器
25	_U2TXInterrupt	_AltU2TXInterrupt	UART2TX Uart 2 发送器
26	_SPI2Interrupt	_AltSPI2Interrupt	SPI2 串行外设接口 2
27	_C1Interrupt	_AltC1Interrupt	CAN1 组合 IRQ
28	_IC3Interrupt	_AltIC3Interrupt	IC3 输入捕捉 3
29	_IC4Interrupt	_AltIC4Interrupt	IC4 输入捕捉 4
30	_IC5Interrupt	_AltIC5Interrupt	IC5 输入捕捉 5
31	_IC6Interrupt	_AltIC6Interrupt	IC6 输入捕捉 6
32	_OC5Interrupt	_AltOC5Interrupt	OC5 输出比较 5
33	_OC6Interrupt	_AltOC6Interrupt	OC6 输出比较 6
34	_OC7Interrupt	_AltOC7Interrupt	OC7 输出比较 7
35	_OC8Interrupt	_AltOC8Interrupt	OC8 输出比较 8
36	_INT3Interrupt	_AltINT3Interrupt	INT3 外部中断 3
37	_INT4Interrupt	_AltINT4Interrupt	INT4 外部中断 4
38	_C2Interrupt	_AltC2Interrupt	CAN2 组合 IRQ
39	_PWMInterrupt	_AltPWMInterrupt	PWM 周期匹配
40	_QEIInterrupt	_AltQEIInterrupt	QEI 位置计数器比较
41	_DCIInterrupt	_AltDCIInterrupt	DCI CODEC 传输完成
42	_LVDInterrupt	_AltLVDInterrupt	PLVD 检测到低电压
43	_FLTAInterrupt	_AltFLTAInterrupt	FLTA MCPWM 故障 A
44	_FLTBInterrupt	_AltFLTBInterrupt	FLTB MCPWM 故障 B
45	_Interrupt45	_AltInterrupt45	保留
46	_Interrupt46	_AltInterrupt46	保留
47	_Interrupt47	_AltInterrupt47	保留
48	_Interrupt48	_AltInterrupt48	保留
49	_Interrupt49	_AltInterrupt49	保留
50	_Interrupt50	_AltInterrupt50	保留
51	_Interrupt51	_AltInterrupt51	保留
52	_Interrupt52	_AltInterrupt52	保留
53	_Interrupt53	_AltInterrupt53	保留

表 **7-2**： 中断向量——**dsPIC30F DSC** （**SMPS**）

IRQ#	主向量名	备用向量名	向量函数
N/A	_ReservedTrap0	_AltReservedTrap0	保留
N/A	_OscillatorFail	_AltOscillatorFail	振荡器故障陷阱
N/A	_AddressError	_AltAddressError	地址错误陷阱
N/A	_StackError	_AltStackError	堆栈错误陷阱
N/A	_MathError	_AltMathError	数学错误陷阱
N/A	_ReservedTrap5	_AltReservedTrap5	保留
N/A	_ReservedTrap6	_AltReservedTrap6	保留
N/A	_ReservedTrap7	_AltReservedTrap7	保留

表 7-2: 中断向量——dsPIC30F DSC （SMPS）（续）

IRQ#	主向量名	备用向量名	向量函数
0	_INT0Interrupt	_AltINT0Interrupt	INT0 外部中断 0
1	_IC1Interrupt	_AltIC1Interrupt	IC1 输入捕捉 1
2	_OC1Interrupt	_AltOC1Interrupt	OC1 输出比较 1
3	_T1Interrupt	_AltT1Interrupt	TMR1 Timer 1 超时
4	_Interrupt4	_AltInterrupt4	保留
5	_OC2Interrupt	_AltOC2Interrupt	OC2 输出比较 2
6	_T2Interrupt	_AltT2Interrupt	TMR2 Timer 2 超时
7	_T3Interrupt	_AltT3Interrupt	TMR3 Timer 3 超时
8	_SPI1Interrupt	_AltSPI1Interrupt	SPI1 串行外设接口 1
9	_U1RXInterrupt	_AltU1RXInterrupt	UART1RX Uart 1 接收器
10	_U1TXInterrupt	_AltU1TXInterrupt	UART1TX Uart 1 发送器
11	_ADCInterrupt	_AltADCInterrupt	ADC 转换完成
12	_NVMInterrupt	_AltNVMInterrupt	NMM NVM 写完成
13	_I2CInterrupt	_AltI2CInterrupt	I^2C™ 中断
14	_I2CErrInterrupt	_AltI2CErrInterrupt	I^2C™ 错误中断
15	_Interrupt15	_AltInterrupt15	保留
16	_INT1Interrupt	_AltINT1Interrupt	INT1 外部中断 1
17	_INT2Interrupt	_AltINT2Interrupt	INT2 外部中断 2
18	_PWMSpEvent MatchInterrupt	_AltPWMSpEvent MatchInterrupt	PWM 特殊事件中断
19	_PWM1Interrupt	_AltPWM1Interrupt	PWM 周期匹配 1
20	_PWM2Interrupt	_AltPWM2Interrupt	PWM 周期匹配 2
21	_PWM3Interrupt	_AltPWM3Interrupt	PWM 周期匹配 3
22	_PWM4Interrupt	_AltPWM4Interrupt	PWM 周期匹配 4
23	_Interrupt23	_AltInterrupt23	保留
24	_Interrupt24	_AltInterrupt24	保留
25	_Interrupt25	_AltInterrupt25	保留
26	_Interrupt26	_AltInterrupt26	保留
27	_Interrupt27	_AltInterrupt27	保留
28	_Interrupt28	_AltInterrupt28	保留
29	_CMP1Interrupt	_AltCMP1Interrupt	模拟比较器中断 1
30	_CMP2Interrupt	_AltCMP2Interrupt	模拟比较器中断 2
31	_CMP3Interrupt	_AltCMP3Interrupt	模拟比较器中断 3
32	_CMP4Interrupt	_AltCMP4Interrupt	模拟比较器中断 4
33	_Interrupt33	_AltInterrupt33	保留
34	_Interrupt34	_AltInterrupt34	保留
35	_Interrupt35	_AltInterrupt35	保留
36	_Interrupt36	_AltInterrupt36	保留
37	_Interrupt37	_AltInterrupt37	保留
38	_Interrupt38	_AltInterrupt38	保留
39	_Interrupt39	_AltInterrupt39	保留
40	_Interrupt40	_AltInterrupt40	保留
41	_Interrupt41	_AltInterrupt41	保留
42	_Interrupt42	_AltInterrupt42	保留
43	_Interrupt43	_AltInterrupt43	保留

表 **7-2**： 中断向量——**dsPIC30F DSC （SMPS）**（续）

IRQ#	主向量名	备用向量名	向量函数
44	_Interrupt44	_AltInterrupt44	保留
45	_Interrupt45	_AltInterrupt45	保留
46	_Interrupt46	_AltInterrupt46	保留
47	_Interrupt47	_AltInterrupt47	保留
48	_Interrupt48	_AltInterrupt48	保留
49	_Interrupt49	_AltInterrupt49	保留
50	_Interrupt50	_AltInterrupt50	保留
51	_Interrupt51	_AltInterrupt51	保留
52	_Interrupt52	_AltInterrupt52	保留
53	_Interrupt53	_AltInterrupt53	保留

表 **7-3**： 中断向量——**PIC24F MCU**

IRQ#	主向量名	备用向量名	向量函数
N/A	_ReservedTrap0	_AltReservedTrap0	保留
N/A	_OscillatorFail	_AltOscillatorFail	振荡器故障陷阱
N/A	_AddressError	_AltAddressError	地址错误陷阱
N/A	_StackError	_AltStackError	堆栈错误陷阱
N/A	_MathError	_AltMathError	数学错误陷阱
N/A	_ReservedTrap5	_AltReservedTrap5	保留
N/A	_ReservedTrap6	_AltReservedTrap6	保留
N/A	_ReservedTrap7	_AltReservedTrap7	保留
0	_INT0Interrupt	_AltINT0Interrupt	INT0 外部中断 0
1	_IC1Interrupt	_AltIC1Interrupt	IC1 输入捕捉 1
2	_OC1Interrupt	_AltOC1Interrupt	OC1 输出比较 1
3	_T1Interrupt	_AltT1Interrupt	TMR1 Timer 1 超时
4	_Interrupt4	_AltInterrupt4	保留
5	_IC2Interrupt	_AltIC2Interrupt	IC2 输入捕捉 2
6	_OC2Interrupt	_AltOC2Interrupt	OC2 输出比较 2
7	_T2Interrupt	_AltT2Interrupt	TMR2 Timer 2 超时
8	_T3Interrupt	_AltT3Interrupt	TMR3 Timer 3 超时
9	_SPI1ErrInterrupt	_AltSPI1ErrInterrupt	SPI1 错误中断
10	_SPI1Interrupt	_AltSPI1Interrupt	SPI1 传输完成中断
11	_U1RXInterrupt	_AltU1RXInterrupt	UART1RX Uart 1 接收器
12	_U1TXInterrupt	_AltU1TXInterrupt	UART1TX Uart 1 发送器
13	_ADC1Interrupt	_AltADC1Interrupt	ADC 1 转换完成
14	_Interrupt14	_AltInterrupt14	保留
15	_Interrupt15	_AltInterrupt15	保留
16	_SI2C1Interrupt	_AltSI2C1Interrupt	从 I^2C™ 中断 1
17	_MI2C1Interrupt	_AltMI2C1Interrupt	主 I^2C™ 中断 1
18	_CompInterrupt	_AltCompInterrupt	比较器中断
19	_CNInterrupt	_AltCNInterrupt	CN 输入变化中断
20	_INT1Interrupt	_AltINT1Interrupt	INT1 外部中断 1
21	_Interrupt21	_AltInterrupt21	保留
22	_Interrupt22	_AltInterrupt22	保留
23	_Interrupt23	_AltInterrupt23	保留

表 7-3: 中断向量——PIC24F MCU （续）

IRQ#	主向量名	备用向量名	向量函数
24	_Interrupt24	_AltInterrupt24	保留
25	_OC3Interrupt	_AltOC3Interrupt	OC3 输出比较 3
26	_OC4Interrupt	_AltOC4Interrupt	OC4 输出比较 4
27	_T4Interrupt	_AltT4Interrupt	TMR4 Timer 4 超时
28	_T5Interrupt	_AltT5Interrupt	TMR5 Timer 5 超时
29	_INT2Interrupt	_AltINT2Interrupt	INT2 外部中断 2
30	_U2RXInterrupt	_AltU2RXInterrupt	UART2RX Uart 2 接收器
31	_U2TXInterrupt	_AltU2TXInterrupt	UART2TX Uart 2 发送器
32	_SPI2ErrInterrupt	_AltSPI2ErrInterrupt	SPI2 错误中断
33	_SPI2Interrupt	_AltSPI2Interrupt	SPI2 传输完成中断
34	_Interrupt34	_AltInterrupt34	保留
35	_Interrupt35	_AltInterrupt35	保留
36	_Interrupt36	_AltInterrupt36	保留
37	_IC3Interrupt	_AltIC3Interrupt	IC3 输入捕捉 3
38	_IC4Interrupt	_AltIC4Interrupt	IC4 输入捕捉 4
39	_IC5Interrupt	_AltIC5Interrupt	IC5 输入捕捉 5
40	_Interrupt40	_AltInterrupt40	保留
41	_OC5Interrupt	_AltOC5Interrupt	OC5 输出比较 5
42	_Interrupt42	_AltInterrupt42	保留
43	_Interrupt43	_AltInterrupt43	保留
44	_Interrupt44	_AltInterrupt44	保留
45	_PMPInterrupt	_AltPMPInterrupt	并行主端口中断
46	_Interrupt46	_AltInterrupt46	保留
47	_Interrupt47	_AltInterrupt47	保留
48	_Interrupt48	_AltInterrupt48	保留
49	_SI2C2Interrupt	_AltSI2C2Interrupt	从 I²C™ 中断 2
50	_MI2C2Interrupt	_AltMI2C2Interrupt	主 I²C™ 中断 2
51	_Interrupt51	_AltInterrupt51	保留
52	_Interrupt52	_AltInterrupt52	保留
53	_INT3Interrupt	_AltINT3Interrupt	INT3 外部中断 3
54	_INT4Interrupt	_AltINT4Interrupt	INT4 外部中断 4
55	_Interrupt55	_AltInterrupt55	保留
56	_Interrupt56	_AltInterrupt56	保留
57	_Interrupt57	_AltInterrupt57	保留
58	_Interrupt58	_AltInterrupt58	保留
59	_Interrupt59	_AltInterrupt59	保留
60	_Interrupt60	_AltInterrupt60	保留
61	_Interrupt61	_AltInterrupt61	保留
62	_RTCCInterrupt	_AltRTCCInterrupt	实时时钟和日历
63	_Interrupt63	_AltInterrupt63	保留
64	_Interrupt64	_AltInterrupt64	保留
65	_U1EInterrupt	_AltU1EInterrupt	UART1 错误中断
66	_U2EInterrupt	_AltU2EInterrupt	UART2 错误中断
67	_CRCInterrupt	_AltCRCInterrupt	循环冗余校验
68	_Interrupt68	_AltInterrupt68	保留

表 **7-3：** 中断向量——**PIC24F MCU** （续）

IRQ#	主向量名	备用向量名	向量函数
69	_Interrupt69	_AltInterrupt69	保留
70	_Interrupt70	_AltInterrupt70	保留
71	_Interrupt71	_AltInterrupt71	保留
72	_Interrupt72	_AltInterrupt72	保留
73	_Interrupt73	_AltInterrupt73	保留
74	_Interrupt74	_AltInterrupt74	保留
75	_Interrupt75	_AltInterrupt75	保留
76	_Interrupt76	_AltInterrupt76	保留
77	_Interrupt77	_AltInterrupt77	保留
78	_Interrupt78	_AltInterrupt78	保留
79	_Interrupt79	_AltInterrupt79	保留
80	_Interrupt80	_AltInterrupt80	保留
81	_Interrupt81	_AltInterrupt81	保留
82	_Interrupt82	_AltInterrupt82	保留
83	_Interrupt83	_AltInterrupt83	保留
84	_Interrupt84	_AltInterrupt84	保留
85	_Interrupt85	_AltInterrupt85	保留
86	_Interrupt86	_AltInterrupt86	保留
87	_Interrupt87	_AltInterrupt87	保留
88	_Interrupt88	_AltInterrupt88	保留
89	_Interrupt89	_AltInterrupt89	保留
90	_Interrupt90	_AltInterrupt90	保留
91	_Interrupt91	_AltInterrupt91	保留
92	_Interrupt92	_AltInterrupt92	保留
93	_Interrupt93	_AltInterrupt93	保留
94	_Interrupt94	_AltInterrupt94	保留
95	_Interrupt95	_AltInterrupt95	保留
96	_Interrupt96	_AltInterrupt96	保留
97	_Interrupt97	_AltInterrupt97	保留
98	_Interrupt98	_AltInterrupt98	保留
99	_Interrupt99	_AltInterrupt99	保留
100	_Interrupt100	_AltInterrupt100	保留
101	_Interrupt101	_AltInterrupt101	保留
102	_Interrupt102	_AltInterrupt102	保留
103	_Interrupt103	_AltInterrupt103	保留
104	_Interrupt104	_AltInterrupt104	保留
105	_Interrupt105	_AltInterrupt105	保留
106	_Interrupt106	_AltInterrupt106	保留
107	_Interrupt107	_AltInterrupt107	保留
108	_Interrupt108	_AltInterrupt108	保留
109	_Interrupt109	_AltInterrupt109	保留
110	_Interrupt110	_AltInterrupt110	保留
111	_Interrupt111	_AltInterrupt111	保留
112	_Interrupt112	_AltInterrupt112	保留
113	_Interrupt113	_AltInterrupt113	保留

表 7-3: 中断向量——PIC24F MCU （续）

IRQ#	主向量名	备用向量名	向量函数
114	_Interrupt114	_AltInterrupt114	保留
115	_Interrupt115	_AltInterrupt115	保留
116	_Interrupt116	_AltInterrupt116	保留
117	_Interrupt117	_AltInterrupt117	保留

表 7-4: 中断向量——dsPIC33F DSC/PIC24H MCU

IRQ#	主向量名	备用向量名	向量函数
N/A	_ReservedTrap0	_AltReservedTrap0	保留
N/A	_OscillatorFail	_AltOscillatorFail	振荡器故障陷阱
N/A	_AddressError	_AltAddressError	地址错误陷阱
N/A	_StackError	_AltStackError	堆栈错误陷阱
N/A	_MathError	_AltMathError	数学错误陷阱
N/A	_DMACError	_AltDMACError	DMA 冲突错误陷阱
N/A	_ReservedTrap6	_AltReservedTrap6	保留
N/A	_ReservedTrap7	_AltReservedTrap7	保留
0	_INT0Interrupt	_AltINT0Interrupt	INT0 外部中断 0
1	_IC1Interrupt	_AltIC1Interrupt	IC1 输入捕捉 1
2	_OC1Interrupt	_AltOC1Interrupt	OC1 输出比较 1
3	_T1Interrupt	_AltT1Interrupt	TMR1 Timer 1 超时
4	_DMA0Interrupt	_AltDMA0Interrupt	DMA 0 中断
5	_IC2Interrupt	_AltIC2Interrupt	IC2 输入捕捉 2
6	_OC2Interrupt	_AltOC2Interrupt	OC2 输出比较 2
7	_T2Interrupt	_AltT2Interrupt	TMR2 Timer 2 超时
8	_T3Interrupt	_AltT3Interrupt	TMR3 Timer 3 超时
9	_SPI1ErrInterrupt	_AltSPI1ErrInterrupt	SPI1 错误中断
10	_SPI1Interrupt	_AltSPI1Interrupt	SPI1 传输完成中断
11	_U1RXInterrupt	_AltU1RXInterrupt	UART1RX Uart 1 接收器
12	_U1TXInterrupt	_AltU1TXInterrupt	UART1TX Uart 1 发送器
13	_ADC1Interrupt	_AltADC1Interrupt	ADC 1 转换完成
14	_DMA1Interrupt	_AltDMA1Interrupt	DMA 1 中断
15	_Interrupt15	_AltInterrupt15	保留
16	_SI2C1Interrupt	_AltSI2C1Interrupt	从 I^2C™ 中断 1
17	_MI2C1Interrupt	_AltMI2C1Interrupt	主 I^2C™ 中断 1
18	_Interrupt18	_AltInterrupt18	保留
19	_CNInterrupt	_AltCNInterrupt	CN 输入变化中断
20	_INT1Interrupt	_AltINT1Interrupt	INT1 外部中断 1
21	_ADC2Interrupt	_AltADC2Interrupt	ADC 2 转换完成
22	_IC7Interrupt	_AltIC7Interrupt	IC7 输入捕捉 7
23	_IC8Interrupt	_AltIC8Interrupt	IC8 输入捕捉 8
24	_DMA2Interrupt	_AltDMA2Interrupt	DMA 2 中断
25	_OC3Interrupt	_AltOC3Interrupt	OC3 输出比较 3
26	_OC4Interrupt	_AltOC4Interrupt	OC4 输出比较 4
27	_T4Interrupt	_AltT4Interrupt	TMR4 Timer 4 超时
28	_T5Interrupt	_AltT5Interrupt	TMR5 Timer 5 超时
29	_INT2Interrupt	_AltINT2Interrupt	INT2 外部中断 2

表 **7-4**：中断向量——**dsPIC33F DSC/PIC24H MCU** （续）

IRQ#	主向量名	备用向量名	向量函数
30	_U2RXInterrupt	_AltU2RXInterrupt	UART2RX Uart 2 接收器
31	_U2TXInterrupt	_AltU2TXInterrupt	UART2TX Uart 2 发送器
32	_SPI2ErrInterrupt	_AltSPI2ErrInterrupt	SPI2 错误中断
33	_SPI2Interrupt	_AltSPI2Interrupt	SPI2 传输完成中断
34	_C1RxRdyInterrupt	_AltC1RxRdyInterrupt	CAN1 接收数据就绪
35	_C1Interrupt	_AltC1Interrupt	CAN1 完成中断
36	_DMA3Interrupt	_AltDMA3Interrupt	DMA 3 中断
37	_IC3Interrupt	_AltIC3Interrupt	IC3 输入捕捉 3
38	_IC4Interrupt	_AltIC4Interrupt	IC4 输入捕捉 4
39	_IC5Interrupt	_AltIC5Interrupt	IC5 输入捕捉 5
40	_IC6Interrupt	_AltIC6Interrupt	IC6 输入捕捉 6
41	_OC5Interrupt	_AltOC5Interrupt	OC5 输出比较 5
42	_OC6Interrupt	_AltOC6Interrupt	OC6 输出比较 6
43	_OC7Interrupt	_AltOC7Interrupt	OC7 输出比较 7
44	_OC8Interrupt	_AltOC8Interrupt	OC8 输出比较 8
45	_Interrupt45	_AltInterrupt45	保留
46	_DMA4Interrupt	_AltDMA4Interrupt	DMA 4 中断
47	_T6Interrupt	_AltT6Interrupt	TMR6 Timer 6 超时
48	_T7Interrupt	_AltT7Interrupt	TMR7 Timer 7 超时
49	_SI2C2Interrupt	_AltSI2C2Interrupt	从 I^2C™ 中断 2
50	_MI2C2Interrupt	_AltMI2C2Interrupt	主 I^2C™ 中断 2
51	_T8Interrupt	_AltT8Interrupt	TMR8 Timer 8 超时
52	_T9Interrupt	_AltT9Interrupt	TMR9 Timer 9 超时
53	_INT3Interrupt	_AltINT3Interrupt	INT3 外部中断 3
54	_INT4Interrupt	_AltINT4Interrupt	INT4 外部中断 4
55	_C2RxRdyInterrupt	_AltC2RxRdyInterrupt	CAN2 接收数据就绪
56	_C2Interrupt	_AltC2Interrupt	CAN2 完成中断
57	_PWMInterrupt	_AltPWMInterrupt	PWM 周期匹配
58	_QEIInterrupt	_AltQEIInterrupt	QEI 位置计数器比较
59	_DCIErrInterrupt	_AltDCIErrInterrupt	DCI CODEC 错误中断
60	_DCIInterrupt	_AltDCIInterrupt	DCI CODEC 传输完成
61	_DMA5Interrupt	_AltDMA5Interrupt	DMA 通道 5 中断
62	_Interrupt62	_AltInterrupt62	保留
63	_FLTAInterrupt	_AltFLTAInterrupt	FLTA MCPWM 故障 A
64	_FLTBInterrupt	_AltFLTBInterrupt	FLTB MCPWM 故障 B
65	_U1ErrInterrupt	_AltU1ErrInterrupt	UART1 错误中断
66	_U2ErrInterrupt	_AltU2ErrInterrupt	UART2 错误中断
67	_Interrupt67	_AltInterrupt67	保留
68	_DMA6Interrupt	_AltDMA6Interrupt	DMA 通道 6 中断
69	_DMA7Interrupt	_AltDMA7Interrupt	DMA 通道 7 中断
70	_C1TxReqInterrupt	_AltC1TxReqInterrupt	CAN1 发送数据请求
71	_C2TxReqInterrupt	_AltC2TxReqInterrupt	CAN2 发送数据请求
72	_Interrupt72	_AltInterrupt72	保留
73	_Interrupt73	_AltInterrupt73	保留
74	_Interrupt74	_AltInterrupt74	保留

表 7-4: 中断向量——**dsPIC33F DSC/PIC24H MCU** （续）

IRQ#	主向量名	备用向量名	向量函数
75	_Interrupt75	_AltInterrupt75	保留
76	_Interrupt76	_AltInterrupt76	保留
77	_Interrupt77	_AltInterrupt77	保留
78	_Interrupt78	_AltInterrupt78	保留
79	_Interrupt79	_AltInterrupt79	保留
80	_Interrupt80	_AltInterrupt80	保留
81	_Interrupt81	_AltInterrupt81	保留
82	_Interrupt82	_AltInterrupt82	保留
83	_Interrupt83	_AltInterrupt83	保留
84	_Interrupt84	_AltInterrupt84	保留
85	_Interrupt85	_AltInterrupt85	保留
86	_Interrupt86	_AltInterrupt86	保留
87	_Interrupt87	_AltInterrupt87	保留
88	_Interrupt88	_AltInterrupt88	保留
89	_Interrupt89	_AltInterrupt89	保留
90	_Interrupt90	_AltInterrupt90	保留
91	_Interrupt91	_AltInterrupt91	保留
92	_Interrupt92	_AltInterrupt92	保留
93	_Interrupt93	_AltInterrupt93	保留
94	_Interrupt94	_AltInterrupt94	保留
95	_Interrupt95	_AltInterrupt95	保留
96	_Interrupt96	_AltInterrupt96	保留
97	_Interrupt97	_AltInterrupt97	保留
98	_Interrupt98	_AltInterrupt98	保留
99	_Interrupt99	_AltInterrupt99	保留
100	_Interrupt100	_AltInterrupt100	保留
101	_Interrupt101	_AltInterrupt101	保留
102	_Interrupt102	_AltInterrupt102	保留
103	_Interrupt103	_AltInterrupt103	保留
104	_Interrupt104	_AltInterrupt104	保留
105	_Interrupt105	_AltInterrupt105	保留
106	_Interrupt106	_AltInterrupt106	保留
107	_Interrupt107	_AltInterrupt107	保留
108	_Interrupt108	_AltInterrupt108	保留
109	_Interrupt109	_AltInterrupt109	保留
110	_Interrupt110	_AltInterrupt110	保留
111	_Interrupt111	_AltInterrupt111	保留
112	_Interrupt112	_AltInterrupt112	保留
113	_Interrupt113	_AltInterrupt113	保留
114	_Interrupt114	_AltInterrupt114	保留
115	_Interrupt115	_AltInterrupt115	保留
116	_Interrupt116	_AltInterrupt116	保留
117	_Interrupt117	_AltInterrupt117	保留

为响应一个中断，必须将函数的地址填充到一个向量表的恰当地址，且函数必须保护它使用的任何系统资源。函数必须使用 RETFIE 处理器指令返回到前台任务。中断函数可用 C 编写。当将一个 C 函数指定为中断处理函数时，编译器将保护编译器使用的所有系统资源，并使用恰当的指令从函数返回。编译器可以可选地用中断函数的地址填充中断向量表。

为使编译器填充指向中断函数的中断向量，按照前表命名函数。例如，如果定义了下列函数，堆栈错误向量将自动被填充：

```
void __attribute__((__interrupt__)) _StackError(void);
```

注意下划线是放在前面的。类似地，如果定义了下面的函数，备用堆栈错误向量将自动被填充：

```
void __attribute__((__interrupt__)) _AltStackError(void);
```

同样要注意下划线是放在前面的。

对于没有指定处理函数的所有中断向量，将自动填充一个默认的中断处理函数。默认的中断处理函数由链接器提供，仅复位器件。应用程序也可通过用名字 _DefaultInterrupt 声明一个中断函数来提供默认的中断处理函数。

每个表中的最后九个中断向量没有预定义的硬件函数。可通过使用前表中给出的名字，或者更适合应用的名字，填充这些向量，同时，仍使用 interrupt 属性的 irq 或 altirq 参数填充适当的向量入口。例如，为指定一个函数使用主中断向量 52，使用下面的语句：

```
void __attribute__((__interrupt__(__irq__(52)))) MyIRQ(void);
```

类似地，指定一个函数使用备用中断向量 52，使用下面的语句：

```
void __attribute__((__interrupt__(__altirq__(52)))) MyAltIRQ(void);
```

irq/altirq号可以为中断请求编号 45至53中的一个。如果使用了interrupt属性的irq 参数，编译器将生成外部符号名 __Interrupt*n*，其中 n 为向量号。因此， C 标识符 _Interrupt45 到 _Interrupt53 被编译器保留。同样，如果使用了 interrupt 属性的 altirq 参数，编译器将生成外部符号 名 __AltInterrupt*n*，其中 n 为向量编号。因此， C 标识符 _AltInterrupt45 到 _AltInterrupt53 被编译器保留。

7.5 中断服务程序现场保护

中断，就其本质来说，在不可预测的时刻发生。因此，被中断的代码必须能以与中断发生时相同的机器状态继续执行。

为正确处理中断返回，中断函数的设置 （prologue）代码自动在堆栈中保护编译器管理的工作寄存器和特殊功能寄存器，以便在 ISR 末尾恢复这些寄存器内容。可使用 `interrupt` 属性的可选 `save` 参数指定要保护和恢复的其他变量和特殊功能寄存器。

在某些应用中，可能需要在中断服务程序中，在编译器生成的函数 prologue 前插入汇编语句。例如，在中断服务程序的入口可能需要递增一个信号。可以这样来实现：

```
void
__attribute__((__interrupt__(__preprologue__("inc _semaphore"))))
  isr0(void);
```

7.6 中断响应时间

有两个因素影响中断源发生到执行 ISR 代码第一条指令之间的周期数。这两个因素是：

- **处理器处理中断时间** — 处理器识别中断并跳转到中断向量第一个地址的时间。这个值与具体器件和所使用中断源有关，为确定这个值的大小，请参考相应器件的数据手册。
- **ISR 代码** — MPLAB C30 在 ISR 中保存它使用的寄存器，这包括工作寄存器和 RCOUNT 特殊功能寄存器。而且，如果 ISR 调用一个普通的函数，编译器要保存所有的工作寄存器和 RCOUNT，即使在 ISR 中没有显式使用这些寄存器。必须要保存这些寄存器，因为一般来说，编译器不知道被调用函数使用了哪些资源。

7.7 中断嵌套

dsPIC30F 器件支持中断嵌套。由于在 ISR 中将处理器资源保存在堆栈中，对嵌套 ISR 的编码与非嵌套中断的编码相同。通过清零 INTCON1 寄存器中的 NSTDIS 位 （嵌套中断禁止位）来使能中断嵌套。注意这是默认设置，因为 dsPIC30F 器件在复位时是使能嵌套中断的。在中断优先级控制寄存器 （IPC*n*）中，为每个中断源分配了一个优先级。如果有一个处于等待状态的中断请求 （IRQ），其优先级等于或高于处理器状态寄存器 （ST 寄存器中的 CPUPRI 字段）中的当前处理器优先级，处理器将响应中断。

7.8 允许 / 禁止中断

可单独禁止或允许每个中断源。每个 IRQ 的都有一个中断允许位位于中断允许控制寄存器 （IEC*n*）中。将中断允许位置 1 将允许相应的中断；将中断允许位清零将禁止相应的中断。器件复位时，所有中断允许位都被清零。另外，处理器还有一个禁止中断指令 （DISI），可在指定的指令周期数内禁止所有中断。

注：	陷阱，如地址错误陷阱，不能禁止。只有 IRQ 是可以被禁止的。

可通过行内汇编可在 C 程序中使用 DISI 指令。例如下面的行内汇编语句：

```
__asm__ volatile ("disi #16");
```

将在源程序中这条语句的所在处发出指定的 DISI 指令。采用这种方式使用 DISI 的一个缺点是， C 编程人员不总是能确定 C 编译器如何将 C 源代码翻译为机器指令，因此可能难以确定 DISI 指令的周期数。通过将要保护以免受中断影响的代码放在 DISI 指令对中断的操作之间，可以解决这个问题。 DISI 指令的第一条指令将周期数设置为最大值，第二条指令将周期数设置为零。例如，

```
__asm__ volatile("disi #0x3FFF"); /* disable interrupts */
/* ... protected C code ... */
__asm__ volatile("disi #0x0000"); /* enable interrupts */
```

另一种可选的方案是直接写 DISICNT 寄存器来允许中断。仅当发出 DISI 指令后且 DISICNT 寄存器的内容非零时，才能修改 DISICNT。

```
__asm__ volatile("disi #0x3FFF"); /* disable interrupts */
/* ... protected C code ... */
DISICNT = 0x0000; /* enable interrupts */
```

对于某些应用，可能还需要禁止优先级为 7 的中断，仅能通过修改 COROCON IPL 字段来禁止这些中断。所提供的支持文件中包含一些有用的预处理宏函数，有助于安全地修改 IPL 值。这些宏是：

```
SET_CPU_IPL(ipl)
SET_AND_SAVE_CPU_IPL(save_to, ipl)
RESTORE_CPU_IPL(saved_to)
```

例如，可能希望保护一个代码段使之不受中断的影响。下面的代码将调整当前的 IPL 设置并将 IPL 恢复到先前的值。

```
void foo(void) {
  int current_cpu_ipl;

  SET_AND_SAVE_CPU_IPL(current_cpu_ipl, 7);  /* disable interrupts */
   /* protected code here */
  RESTORE_CPU_IPL(current_cpu_ipl);
}
```

7.9 中断服务程序和一般代码共用存储空间

当修改主代码或低优先级中断服务程序和高优先级中断服务程序中同时使用的同一变量时，一定要小心。当高优先级中断允许时，可中断多指令序列，且当低优先级中断函数生成访问同一变量的多指令读－修改－写序列时可能产生意想不到的结果。因此，嵌入式系统必须实现*原子* （*atomic*）操作来确保干预高优先级中断服务程序不会写低优先级中断服务程序刚刚读取但还未完成写的同一变量。

原子操作指执行时作为一个整体而不可分割的操作——这种操作不能被中断。并非所有的 C 表达式都解释为原子操作，这取决于所使用器件的架构。对于 dsPIC DSC 器件，这样的表达式主要分为如下类型：32 位表达式、浮点算术、除法和对多位位域的操作。其他因素决定是否产生原子操作，如存储模型设置、优化级别和可用资源。

考虑下面的通用表达式：

```
foo = bar op baz;
```

操作符 （op）可以为原子的或非原子的，这由器件的架构决定。无论如何，编译器不会在所有情况下都能生成原子操作——这在很大程度上取决于如下几个因素：

- 适当原子机器指令的可用性
- 资源可用性——特殊寄存器或其他约束
- 优化级别，以及其他影响数据 / 代码分配的选项

在不了解架构的情况下，假设通用表达式需要两次读 （一次读取一个操作数）和一次写 （用来存储结果）是合理的。在中断序列存在的情况下可能会遇到一些困难，这与具体的应用有关。

7.9.1 开发注意要点

下面给出几个示例：

例 7-1： BAR 与 BAZ 必须匹配

如果要求 bar 与 baz 匹配 （即它们被 同时更新），则如果 bar 或 baz 之一可在高优先级中断表达式中更新，就可能会有危险。下面给出了一些流程序列：

1. 安全 读 bar
读 baz
执行操作
将结果写回 foo
2. 不安全 读 bar
中断修改 baz
读 baz
执行操作
将结果写回 foo

3. 安全 读 bar

读 baz

中断修改 bar 或 baz

执行操作

将结果写回 foo

第一个序列是安全的，因为在表达式范围内没有中断。第二个序列是不安全的，因为应用要求 bar 和 baz 必须同时更新。第三个序列有可能安全， foo 可能有一个旧值，但这个值和表达式开头时的数据一致。

例 7-2： FOO、 BAR 和 BAZ 的类型

另一个要注意的要点取决于 foo、 bar 和 baz 的类型。操作 “读 bar”、“读 baz” 或 “将结果写回 foo”可能不是原子的，这取决于目标处理器的架构。例如， dsPIC DSC 器件可通过 1 条（原子）指令读或写一个 8 位、16 位或 32 位量。但是，对于 32 位量可能需要两条指令，这取决于指令的选择 （而指令选择取决于优化和存储模型的设置）。假设其类型为 long，且编译器不能选择原子操作来访问数据，那么访问变为：

读 bar 低位字

读 bar 高位字

读 baz 低位字

读 baz 高位字

（对低位字和高位字）执行操作

执行操作

将结果的低位字写回 foo

将结果的高位字写回 foo

现在，更新 bar 或 baz 导致异常数据的可能性更大了。

例 7-3： 位域

第三个要注意的方面是位域。 C 允许按位分配存储空间，但并未定义任何位操作。从最纯粹的意义上来讲，对一个位进行的任何操作都将视为对位域的底层类型的操作，且都需要某些操作来从 bar 和 baz 中抽取字段或将字段插入到 foo 中。要注意的重要一点是，写入 foo 所在位域的任何部分的被中断子程序可能容易被破坏 （这也取决于指令结构、优化级别和存储设置）。这一点在一个操作数也是目标操作数时尤其明显。

dsPIC DSC 指令集可原子地对一个位进行操作。编译器可选择这种指令，这由优化级别、存储设置和资源可用性决定。

例 7-4： 将存储值暂存在寄存器中

要注意的最后一点，编译器可能选择将存储值暂存在寄存器中。这种变量通常称为寄存器变量，尤其容易受到中断的破坏，即使在对变量进行的操作不被中断的情况下亦是如此。要确保将中断服务程序和可中断函数之间共用的存储资源指定为 volatile。这将通知编译器对存储位置的更新可与顺序代码序列不同步。这样虽不能防止非原子操作的影响，但却很重要。

7.9.2 开发方案

下面提供了几种方法来消除潜在的危险：

- 软件系统的设计要确保不发生冲突事件。不要使中断服务程序和其他函数之间共用存储空间。中断服务程序要尽可能简单，具体工作由主代码完成。
- 当共用存储空间时要尽量小心，避免共用包含多个位的位域。
- 要像保护关键代码段一样，防止共用存储空间的非原子更新受到中断的影响。下面的宏可用来实现这一点：

```
#define INTERRUPT_PROTECT(x) {              \
   char saved_ipl;                          \
                                            \
   SET_AND_SAVE_CPU_IPL(saved_ipl,7);       \
   x;                                       \
   RESTORE_CPU_IPL(saved_ipl); } (void) 0;
```

  这个宏通过将当前的中断优先级增加为 7，执行需要的语句，然后恢复到原来的优先级来禁止中断。

7.9.3 应用示例

下面的示例着重说明了本节 （第 7.9 节）中讲述的要点：

```
void __attribute__((interrupt))
     HigherPriorityInterrupt(void) {
      /* User Code Here */
      LATGbits.LATG15 = 1;  /* Set LATG bit 15 */
      IPC0bits.INT0IP = 2;  /* Set Interrupt 0
                               priority (multiple
                               bits involved) to 2 */
       }

int main(void) {
    /* More User Code */
    LATGbits.LATG10 ^= 1;  /* Potential HAZARD -
                              First reads LATG into a W reg,
                              implements XOR operation,
                              then writes result to LATG */

    LATG = 0x1238;         /* No problem, this is a write
                              only assignment operation */

    LATGbits.LATG5 = 1;    /* No problem likely,
                              this is an assignment of a
                              single bit and will use a single
                              instruction bit set operation */

    LATGbits.LATG2 = 0;    /* No problem likely,
                              single instruction bit clear
                              operation probably used */

    LATG += 0x0001;        /* Potential HAZARD -
                              First reads LATG into a W reg,
                              implements add operation,
                              then writes result to LATG */

    IPC0bits.T1IP = 5;     /* HAZARD -
                              Assigning a multiple bitfield
                              can generate a multiple
                              instruction sequence */

}
```

使用前面提供的宏 INTERRUPT_PROTECT 可防止一条语句受到中断的影响。例如：

```
INTERRUPT_PROTECT(LATGbits.LATG15 ^= 1); /* Not interruptible by
                                            level 1-7 interrupt
                                            requests and safe
                                            at any optimization
                                            level */
```

注：



第 8 章　汇编语言和 C 模块混合编程

8.1　简介

本章讲述如何混合使用汇编语言和 C 模块。给出了在汇编代码中使用 C 变量和函数的例子，以及在 C 中使用汇编变量和函数的例子。

8.2　主要内容

本章主要讨论的内容有：

- **在汇编语言中使用 C 变量和 C 函数** — 可对独立的汇编模块进行汇编，然后于编译后的 C 模块相链接。
- **使用行内汇编** — 汇编指令可直接嵌入到 C 代码中。行内汇编既支持简单的 （不带参数的）汇编语句，也支持扩展的 （带参数的）汇编语句，其中 C 变量可作为汇编指令的操作数访问。

8.3　在汇编语言中使用 C 变量和 C 函数

下面的指导原则指明了独立的汇编模块如何与 C 模块接口。

- 遵循**第 4.13 节 "寄存器约定"**所述的寄存器约定。尤其是，寄存器 W0-W7 用于参数传递。汇编函数将接收参数，并通过这些寄存器将参数传递到被调用函数。
- 不在中断处理期间调用的函数必须保护寄存器 W8-W15。即这些寄存器的值必须在被修改前保存并在返回到调用函数前恢复。使用寄存器 W0-W7 时可不恢复其值。
- 中断函数必须保护所有的寄存器。不同于一般的函数调用，中断可能发生在程序执行过程中的任意时刻。当返回到正常的程序执行时，所有寄存器的值必须与中断发生前相同。
- 汇编文件中声明的变量或函数，如果要被任何 C 源文件引用，应该使用汇编伪指令 `.global` 声明为全局的。外部符号前应至少有一个下划线。C 函数 `main` 在汇编中命名为 `_main`，相反，汇编符号 `_do_something` 在 C 中引用为 `do_something`。在汇编文件中使用的未声明符号将视为外部定义的。

下面的示例说明了如何在汇编语言和 C 中使用变量和函数，而与变量和函数原来是在哪里定义的无关。

文件 `ex1.c` 定义了将在汇编文件中使用的 `foo` 和 `cVariable`。 C 文件还说明了如何调用汇编函数 `asmFunction`，以及如何访问汇编定义的变量 `asmVariable`。

例 8-1： C 和汇编混合编程

```
/*
** file: ex1.c
*/
extern unsigned int asmVariable;
extern void asmFunction(void);
unsigned int cVariable;
void foo(void)
{
  asmFunction();
  asmVariable = 0x1234;
}
```

文件 ex2.s 定义了链接应用程序需要使用的 asmFunction 和 asmVariable。汇编文件还说明了如何调用 C 函数 foo，以及如何访问 C 定义的变量 cVariable。

```
;
;  file: ex2.s
;
  .text
  .global _asmFunction
_asmFunction:
  mov #0,w0
  mov w0,_cVariable
  return

  .global _begin
_main:
  call _foo
  return

  .bss
  .global _asmVariable
  .align 2
_asmVariable: .space 2
 .end
```

在 C 文件 ex1.c 中，使用了标准的 extern 关键字声明了对汇编文件中定义的符号的外部引用；注意汇编源文件中的 asmFunction 或 _asmFunction 是一个 void 函数，进行了相应声明。

在汇编文件 ex1.s 中，通过使用 .global 汇编伪指令，使符号 _asmFunction、_main 和 _asmVariable 全局可见，并可被任何其他源文件访问。 仅引用了符号 _main，未进行声明；因此，汇编器将其视为外部引用。

下面的 MPLAB C30 示例展示了如何调用带两个参数的汇编函数。 call1.c 中的 C 函数 main 调用 call2.s 中带两个参数的 asmFunction。

例 8-2： 在 C 中调用汇编函数

```
/*
** file: call1.c
*/
extern int asmFunction(int, int);
int x;
void
main(void)
{
 x = asmFunction(0x100, 0x200);
}
```

汇编函数将两个参数相加，并返回结果。

```
;
; file: call2.s
;
  .global _asmFunction
_asmFunction:
  add w0,w1,w0
  return
 .end
```

关于 C 中参数传递的详细描述，请参阅**第 4.12.2 节 "返回值"**。在前面的例子中，两个整型参数通过 W0 和 W1 寄存器传递。整型返回结果通过寄存器 W0 传递。对于更为复杂的参数列表，可能需要更多不同的寄存器，且要注意在编写汇编程序时要遵循指导原则。

8.4 使用行内汇编

在 C 函数中，可使用 asm 语句将一行汇编代码插入到编译器生成的汇编语言中。行内汇编有两种形式：简单的和扩展的。

在**简单**形式中，使用下面的语法写汇编指令：

```
asm ("instruction");
```

其中 *instruction* 是一个有效的汇编语言语法结构。如果在 ANSI C 程序中写行内汇编，使用 __asm__ ，而不是 asm。

> **注：** 只能传递一个字符串到简单形式的行内汇编。

在使用 asm 的**扩展**汇编指令中，使用 C 表达式指定指令的操作数。扩展行内汇编的语法如下：

```
asm("template" [ : [ "constraint"(output-operand) [ , ... ] ]
                 [ : [ "constraint"(input-operand) [ , ... ] ]
                      [ "clobber" [ , ... ] ]
               ]
             ]);
```

必须指定汇编指令 *template*，并为每个操作数指定操作数 *constraint* 字符串。*template* 指定指令助记符，以及操作数的占位符（可选）。*constraint* 字符串指定操作数约束，例如，指定一个操作数必须位于寄存器中（通常是这样的），或者操作数必须为立即值。

MPLAB C30 支持如下约束字母。

表 8-1： MPLAB® C30 支持的约束字母

字母	约束
a	声明 WREG
b	除法支持寄存器 W1
c	乘法支持寄存器 W2
d	通用数据寄存器 W1 - W14
e	非除法支持寄存器 W2 - W14
g	允许任何寄存器、存储区或立即整型操作数，不是一般寄存器的寄存器除外。
i	允许 （具有常数值的）立即整型操作数。这包括其值仅在汇编时已知的符号常量。
r	允许寄存器操作数，前提是寄存器要为一般寄存器。
v	累加器回写寄存器 W13
w	累加器 A - B
x	x 预取寄存器 W8 - W9
y	y 预取寄存器 W10 - W11
z	MAC 预取寄存器 W4 - W7
0, 1, … , 9	允许与指定操作数编号匹配的操作数。如果在同一选择中，数字和字母一起使用，应把数字放在后面。
T	near 或 far 数据操作数。
U	near 数据操作数。

表 8-2： MPLAB® C30 支持的约束修饰符

字母	约束
=	表明这个操作数对于这条指令来说是只写的：先前的值被舍弃并被输出数据代替。
+	表明指令可读写这个操作数。
&	表明这个操作数是一个 *earlyclobber* 操作数，在指令使用完输入操作数之前被修改了。因此，这个操作数不能存放在用作输入操作数或任何存储地址一部分的寄存器中。

例 8-3： 传递 C 变量

这个示例说明了如何使用 swap 指令 （编译器一般不使用这条指令）：

```
asm ("swap %0" : "+r"(var));
```

其中 var 是操作数的 C 表达式，同时为输入操作数和输出操作数。该操作数约束为类型 r，表示为寄存器操作数。 +r 中的 + 表明该操作数既是输入操作数，也是输出操作数。

每个操作数都通过操作数约束字符串，后跟括在括号中的 C 表达式来描述。冒号将汇编模板与第一个输出操作数分隔开，另外一个冒号将最后一个输出操作数和第一个输入操作数分隔开 （如果存在的话）。逗号分隔开各输出操作数和各输入操作数。

如果没有输出操作数，有输入操作数，那么在输出操作数的两边必须有两个连续的逗号。编译器要求输出操作数表达式必须为左值。输入操作数不必为左值。编译器不能检查出操作数是否具有对于正在执行指令合理的数据类型。它不解析汇编指令模板，不知道其含义，或者是否是有效的汇编输入。扩展的 asm 功能经常用于编译器本身不知道其存在的机器指令。如果不能对输出表达式直接寻址 （例如它是一个位域），那么约束必须允许寄存器。在这种情况下， MPLAB C30 将使用寄存器作为 asm 的输出，然后将该寄存器的值存储到输出。如果输出操作数是只写的， MPLAB C30 将假定执行指令之前这些操作数中的值是死值，不需要输出。

例 8-4：　　　　　破坏寄存器

某些指令会破坏特定硬件寄存器的值。为说明这一点，在输入操作数后写第三个逗号，后跟被破坏的硬件寄存器的名字 （作为字符串给出，中间用逗号隔开）。下面是一个例子：

```
asm volatile ("mul.b %0"
: /* no outputs */
: "U" (nvar)
: "w2");
```

在这个例子中，由于指定了 “U” 约束，操作数 nvar 是声明到 near 数据空间的字符变量。如果汇编指令可能修改标志 （条件代码）寄存器，添加 “cc” 到被破坏寄存器的列表。如果汇编指令以不可预估的方式修改存储器，添加 “memory” 到被破坏寄存器的列表。这将使 MPLAB C30 不跨汇编指令保存存储在寄存器中的存储器值。

例 8-5：　　　　　使用多条汇编指令

可以在一个 asm 模板中放多条汇编指令，中间用换行 （写 \n）分隔开。输入操作数和输出操作数的地址保证不会使用任何被破坏的寄存器，所以可以任意多次读和写被破坏的寄存器。下面是在一个模板中有多条指令的例子；它假定子程序 _foo 接受寄存器 W0 和 W1 中的参数：

```
asm ("mov %0,w0\nmov %1,W1\ncall _foo"
: /* no outputs */
: "g" (a), "g" (b)
: "W0", "W1");
```

在这个例子中，约束字符串 “g” 表明一个普通操作数。

例 8-6：　　　　　使用 “&” 来防止破坏输入寄存器

假定在产生输出之前使用了输入， MPLAB C30 可能将输出操作数分配到无关的输入操作数存放的寄存器，除非输出操作数有 & 约束修饰符。如果汇编代码实际上包含多条指令，这个假定可能是错误的。在这种情况下，对每个输出操作数使用 & ，不会与输入操作数重叠。例如，考虑下面的函数：

```
int
exprbad(int a, int b)
{
        int c;

        __asm__("add %1,%2,%0\n sl %0,%1,%0"
                : "=r"(c) : "r"(a), "r"(b));

        return(c);
}
```

这个函数的目的是计算值 (a + b) << a。但是，正如所写的，计算的值可能是或不是这个值。正确的编码是通知编译器在 asm 指令使用完输入操作数之前修改了操作数 c，如下所示：

```
int
exprgood(int a, int b)
{
        int c;

        __asm__("add %1,%2,%0\n sl %0,%1,%0"
                : "=&r"(c) : "r"(a), "r"(b));

        return(c);
}
```

例 8-7： 匹配操作数

当汇编指令有一个读 - 写操作数，或者有一个仅某些位将被修改的操作数时，必须在逻辑上将其功能拆分为两个独立的操作数：一个是输入操作数，一个是只写输出操作数。这两个操作数之间的关系由指明执行指令时两个操作数需要在同一地址的约束表明。可对两个操作数使用相同的或不同的 C 表达式。例如，下面的 add 指令中， bar 作为只读源操作数，foo 作为读 – 写目标操作数：

```
asm ("add %2,%1,%0"
: "=r" (foo)
: "0" (foo), "r" (bar));
```

操作数 1 的约束 “0”表明该操作数必须与操作数 0 占用相同的地址。仅允许在输入操作数的约束中使用数字，并且必须指向输出操作数。约束中只有数字可保证一个操作数将和另一个操作数存放在同一地址。 foo 是两个操作数的值并不足以保证两个操作数在生成的汇编代码中位于相同的地址。下面的代码可以做到这一点：

```
asm ("add %2,%1,%0"
: "=r" (foo)
: "r" (foo), "r" (bar));
```

各种优化和重载可能导致操作数 0 和操作数 1 位于不同的寄存器中。例如，编译器可能在一个寄存器中找到 foo 的值的一个拷贝，将这个寄存器用于操作数 1，但是在另一个寄存器中生成了输出操作数 0 （将其拷贝到 foo 自身地址的后面）。

例 8-8： 指定操作数

还可以使用可在汇编代码模板内引用的符号名来指定输入和输出操作数。这些符号名在约束字符串前的方括号内指定，可使用 %[*name*] （而不是百分号加操作数编号）在汇编代码模板内引用。使用指定的操作数，上面的例子可编码如下：

```
asm ("add %[foo],%[bar],%[foo]"
: [foo] "=r" (foo)
: "0" (foo), [bar] "r" (bar));
```


通过在 asm 后写关键字 volatile，可禁止 asm 指令被删除、明显移动或组合。例如：

```
#define disi(n) \
asm volatile ("disi #%0" \
: /* no outputs */ \
: "i" (n))
```

在这个例子中，约束字母 “i” 表示 disi 指令要求的立即操作数。

例 8-9： 使控制流发生改变

当行内汇编语句会使控制流发生改变时，有一些特殊注意事项。

例如，无法告知编译器行内汇编语句会导致控制流发生改变。控制应该进入汇编语句并始终继续执行下一条语句。

良好的控制流：

```
  asm("call _foo" : /* outputs */
                  : /* inputs */
                  : "w0", "w1", "w2", "w3", "w4", "w5",
                    "w6", "w7");
  /* next statement */
```

这是可接受的，因为调用 foo 后将执行下一条语句。代码告知编译器在这条语句后不使用某些寄存器；这些寄存器指 foo 不保护的寄存器。

错误的控制流：

```
  asm("bra OV, error");
  /* next statement */
  return 0;

  asm("error: ");
  return 1;
```

这是不可接受的，因为编译器将假定执行下一条语句 return 0，而实际上不应该执行这条语句。在这种情况下， asm("error: ") 及其后面的语句将由于执行不到被删除 。请参阅下文关于汇编语句中标号的信息。

可接受的控制流：

```
  asm("cp0 _foo\n\t"
      "bra nz, eek\n\t"
      "; some assembly\n\t"
      "bra eek_end\n\t"
      "eek:\n\t"
      "; more assembly\n"
      "eek_end:" : /* outputs */
                 : /* inputs */
                 : "cc");
  /* next statement */
```

这是可接受的，但不能按预期执行 （即，总是执行下一条语句，而与汇编语句内的任何跳转无关）。请参阅下文关于汇编语句中标号的信息。注意，通过将 cc 标识为被破坏的，代码表明状态标志在这条语句中不再有效。

标号及控制流：

此外，某些优化选项可能会使汇编语句中的标号工作异常。行内汇编器可能使汇编语句内的标号被多次定义。

而且过程聚合工具 （-mpa）不接受局部标号语法。请看下面的示例：

```
inline void foo() {
  asm("do #6, loopend");
  /* some C code */
  asm("loopend: ");
  return;
}
```

这是错误的。首先，汇编语句引入了编译器未知的隐含控制流。其次，如果嵌入了 foo()，标号 loopend 将被多次定义。

解决方案是使用 *《MPLAB® ASM30、 MPLAB LINK30 和实用程序用户指南》*（DS51317D_CN）中所述的局部标号语法，如下例所示：

```
inline void foo() {
  asm("do #6, 0f");
  /* some C code */
  asm("0: ");
  return;
}
```

上面的形式稍微好些，至少它能解决多次定义标号的问题。但是过程聚合工具（-mpa）不接受标号的 0：形式。

例 8-10： 使用要求的寄存器

dsPIC DSC 指令集中的某些指令要求将操作数存放在特定寄存器或寄存器组中。表 8-1 列出了可能适合于满足希望生成的指令的约束的一些约束字母。

如果约束不满足或希望指定汇编语句中使用的特定寄存器，可使用编译器提供的寄存器指定扩展来支持 （并降低将寄存器标识为被破坏的需要），如下面的代码所示。这段代码使用了一条虚构指令，这条指令具有一些非正常的寄存器要求：

```
{ register int in1 asm("w7");
  register int in2 asm("w9");
  register int out1 asm("w13");
  register int out2 asm("w0");

  in1 = some_input1;
  in2 = some_input2;
  __asm__ volatile ("funky_instruction %2,%3,%0; = %1" :
                    /* outputs */ "=r"(out1), "=r"(out2) :
                    /* inputs */ "r"(in1), "r"(in2));
  /* use out1 and out2 in normal C */
}
```

在这个例子中， funky_instruction 具有一个显式输出 out1 和一个隐式输出 out2。两个输出都包含在 asm 模板中 （尽管隐式输出在注释语句中），使得编译器可以正确跟踪寄存器的使用。所示的输入是正常的。否则，使用扩展的寄存器声明语法来指定满足虚构 funky_instruction 约束的特定硬件寄存器。



附录 A 实现定义的操作

A.1 简介

本章讲述 MPLAB C30 实现定义的操作。 C 的 ISO 标准要求各厂商提供关于语言的“实现定义”细节方面的文档。

本章讲述的主要内容有：

- 翻译
- 环境
- 标识符
- 字符
- 整型
- 浮点型
- 数组和指针
- 寄存器
- 结构、联合、枚举和位域
- 限定符
- 声明符
- 语句
- 预处理伪指令
- 库函数
- 信号
- 流和文件
- tmpfile
- errno
- 存储区
- abort
- exit
- getenv
- system
- strerror

A.2 翻译

翻译的实现定义在 ANSI C 标准的 G.3.1 章节中有详细介绍。

是否空白字符 （换行符除外）的每个非空序列都被保留了，或者它被一个空格字符替代？ （ISO 5.1.1.2）

它被一个空格字符替代。

如何标识一个诊断消息？ （ISO 5.1.1.3）

诊断消息是通过用与消息对应的源文件名和行号为前缀来标识的，用冒号 （“：”）分隔开。

是否存在不同种类的消息？ （ISO 5.1.1.3）

是的。

如果是，它们都有哪些？ （ISO 5.1.1.3）

错误消息，禁止产生输出文件；警告消息，不禁止输出文件的生成。

翻译器对于每种消息返回什么状态代码？ （ISO 5.1.1.3）

对于错误消息，返回状态码 1 ；对于警告消息，返回状态码 0。

可以控制诊断的级别吗？

是的。

如果是，那么控制会采用什么形式？

可使用编译器命令行选项来请求或禁止警告消息的产生。

A.3 环境

环境的实现定义在 ANSI C 标准的 G.3.2 章节中有详细介绍。

哪些库函数可供独立的程序使用？ （ISO 5.1.2.1）

标准 C 函数库的所有函数都可使用，只要为环境定制了一小组函数，这在 “运行时函数库” 章节中有详细介绍。

描述一下在独立环境中的程序终止。

如果主函数返回或者调用了函数退出，会在无限循环内执行 `HALT` 指令。这种操作用户是可以自定义的。

描述一下传递到 `main` 函数的参数？ （ISO 5.1.2.2.1）

没有参数传递到 `main` 函数。

判断以下是否是合法的交互设备：（ISO 5.1.2.3）

异步终端 不是

显示器和键盘 不是

程序间连接 不是

其他？请描述 没有

A.4 标识符

标识符的实现定义在 ANSI C 标准的 G.3.3 章节中有详细介绍。

在没有外部链接的标识符中，除 31 个字符外，有多少字符是有效的？（ISO 6.1.2）

所有字符都是有效的。

在有外部链接的标识符中，除 6 个字符外，有多少字符是有效的？（ISO 6.1.2）

所有字符都是有效的。

对于有外部链接的标识符中，是否区分大小写？（ ISO 6.1.2）

是的。

A.5 字符

字符的实现定义在 ANSI C 标准的 G.3.4 章节中有详细介绍。

请详述标准未明确指定的任何源字符和执行字符。（ISO 5.2.1）

无。

列出所列出序列产生的转义序列值。 （ISO 5.2.2）

表 A-1： 转义序列字符及其值

序列	值
\a	7
\b	8
\f	12
\n	10
\r	13
\t	9
\v	11

执行字符集中的一个字符中有多少位？ （ISO 5.2.4.2）

8 位。

源字符集 （字符和字符串）中的元素到执行字符集中元素的映射是什么？（ISO 6.1.3.4）

identity 函数。

无符号说明的 `char` 的等价类型是什么？ （ISO 6.2.1.1）

都可以 （由用户定义）。默认等价类型为 `signed char`。可使用一个编译器命令行选项来使默认类型变为 `unsigned char`。

A.6 整型

整型的实现定义在 ANSI C 标准的 G.3.5 章节中有详细介绍。

下面列出了各种整型的长度及其范围：（ISO 6.1.2.5）

表 A-2： 整型

数据类型	长度（位数）	范围
`char`	8	-128 … 127
`signed char`	8	-128 … 127
`unsigned char`	8	0 … 255
`short`	16	-32768 … 32767
`signed short`	16	-32768 … 32767
`unsigned short`	16	0 … 65535
`int`	16	-32768 … 32767
`signed int`	16	-32768 … 32767
`unsigned int`	16	0 … 65535
`long`	32	-2147483648 … 2147438647
`signed long`	32	-2147483648 … 2147438647
`unsigned long`	32	0 … 4294867295

将一个整型转换为较短的有符号整型，或者将一个无符号整型转换为相同长度的有符号整型，如果不能表示值，结果怎样？（ISO 6.2.1.2）

会丢失有效位。不会发出错误。

对有符号整型进行位操作的结果是什么？（ISO 6.3）

移位操作符保留符号。其他操作符象操作数为无符号整型一样进行操作。

整型除法中余数的符号是什么？（ISO 6.3.5）

+。

负值有符号整型右移的结果是什么？（ISO 6.3.7）

保留符号。

A.7 浮点型

浮点型的实现定义在 ANSI C 标准的 G.3.6 中讲述。

在浮点型的可表示值范围内的浮点型常量定标值，是最相近的可表示值？还是与之相邻且大于它的可表示值？还是与之相邻且小于它的可表示值？（ISO 6.1.3.1）

最相近的可表示值。

下表给出了浮点数的各种类型三长度及其范围：（ISO 6.1.2.5）

表 **A-3**： 浮点型

数据类型	长度（位）	范围
`float`	32	1.175494e-38 … 3.40282346e+38
`double`*	32	1.175494e-38 … 3.40282346e+38
`long double`	64	2.22507385e-308 … 1.79769313e+308

* 如果使用了 `-fno-short-double`， `double` 与 `long double` 等价。

将整型数转化为浮点数，而浮点数不能精确地表示原来的值时，截取方向如何？（ISO 6.2.1.3）

向下截取。

当将一个浮点数转化为较短的浮点数时，截取或舍入方向如何？ （ISO 6.2.1.4）

向下截取或舍入。

A.8 数组和指针

数组和指针的实现定义在 ANSI C 标准的 G.3.7 章节中讲述。

保存数组最大长度所需的整型，即操作符长度 `size_t` 的类型是什么？ （ISO 6.3.3.4， ISO 7.1.1）

`unsigned int`。

将指针转化为整型所需要的整型的长度？ （ISO 6.3.4）

16 位。

将指针强制转换为整型或反之的结果是什么？ （ISO 6.3.4）

映射为 identity 函数。

保存指向同一数组的元素的两个指针之间的区别所需的整型 `ptrdiff_t`？ （ISO 6.3.6， ISO 7.1.1）

`unsigned int`。

A.9 寄存器

寄存器的实现定义在 ANSI C 标准的 G.3.8 章节中讲述。

存储类别修饰符 `register` 实际上在多大程度上影响将对象存储在寄存器中？ （ISO 6.5.1）

如果禁止优化，指定 `register` 存储类别；否则忽略这个修饰符。

A.10 结构、联合、枚举和位域

结构、联合、枚举和位域的实现定义在 ANSI C 标准的 A.6.3.9 和 G.3.9 章节中讲述。

如果联合对象中的一个成员被一个不同类型的成员访问结果如何？（ISO 6.3.2.3）

不进行转换。

描述一下结构的成员的填充和 对齐？（ISO 6.5.2.1）

字符是字节对齐的。所有其他对象是字对齐的。

无符号说明的 `int` 位域的等价类型是什么？（ISO 6.5.2.1）

由用户定义。默认情况下是 `signed int` 位域。可使用一个命令行选项使等价类型变为 `unsigned int` 位域。

int 中位域的分配顺序如何？（ISO 6.5.2.1）

位以自低位向高位的顺序分配。

位域能否跨越存储单元边界？（ISO 6.5.2.1）

能。

选择哪个整型来表示枚举类型的值？（ISO 6.5.2.2）

`int`。

A.11 限定符

限定符的实现定义在 ANSI C 标准的 G.3.10 章节中讲述。

描述一下什么操作构成对具有 `volatile` 限定类型的对象的访问？（ISO 6.5.3）

如果在一个表达式中指定了一个对象，则它已经被访问了。

A.12 声明符

声明符的实现定义在 ANSI C 标准的 G.3.11 章节中讲述。

可修改算术、结构或联合类型的声明符的最大数目？（ISO 6.5.4）

没有限制。

A.13 语句

语句的实现定义在 ANSI C 标准的 G.3.12 章节中讲述。

switch 语句中 case 值的最大值是多少？（ISO 6.6.4.2）

没有限制。

A.14 预处理伪指令

预处理伪指令的实现定义在 ANSI C 标准的 G.3.13 章节中讲述。

控制条件包含的条件表达式中的单字符字符常量的值是否与执行字符集中的同一字符常量相符？ （ISO 6.8.1）

是的。

这种字符常量可以有负值吗？ （ISO 6.8.1）

是的。

可使用什么方法来定位可包含源文件？ （ISO 6.8.2）

预处理器搜索当前目录，然后搜索使用命令行选项指定的目录。

头文件如何标识？其位置如何指定？ （ISO 6.8.2）

头文件通过 `#include` 伪指令标识，括在 < 和 > 分隔符之间，或者 “ 和 ” 分隔符之间。使用命令行选项指定其位置。

可将可包含源文件的名字用引号括起来吗？ （ISO 6.8.2）

是的。

分隔的字符序列和外部源文件名之间的映射是什么？ （ISO 6.8.2）

identity 函数。

描述一下可识别的 #pragma 伪指令的操作。（ISO 6.8.6）

表 A-4： #PRAMA 操作

Pragma	操作
`#pragma code section-name`	命名代码段。
`#pragma code`	将代码段名重设为默认段名 （即 `.text`）。
`#pragma idata section-name`	命名已初始化的数据段。
`#pragma idata`	将已初始化的数据段名重设为其默认值 （即 `.data`）。
`#pragma udata section-name`	命名未初始化的数据段。
`#pragma udata`	将未初始化的数据段名重设为其默认值 （即 `.bss`）。
`#pragma interrupt function-name`	将函数名指定为中断函数。

当没有转换的日期和时间时， __DATE__ 和 __ TIME __ 的定义分别是什么？ （ISO 6.8.8）

不适用。在没有这些函数的环境中不支持编译器。

A.15 库函数

库函数的实现定义在 ANSI C 标准的 G.3.14 章节中讲述。

宏 NULL 扩展到的空指针常量是什么？（ISO 7.1.5）

0。

是如何识别 assert 函数打印的诊断消息的，函数是如何终止的？（ISO 7.2）

assert 函数打印文件名、行号和检测表达式，中间用冒号字符（“:”）分割开。然后调用 abort 函数。

通过 isalnum、isalpha、scntrl、islower、isprint 和 isupper 函数检测哪些字符？（ISO 7.3.1）

表 A-5： IS 函数检测的字符

函数	检测的字符
isalnum	一个字母或数字：isalpha 或 isdigit。
isalpha	一个字母：islower 或 isupper。
iscntrl	五个标准运动控制字符、退格和警告字符之一：\f、\n、\r、\t、\v、\b 和 \a。
islower	“a” 到 “z” 中的一个字母。
isprint	图形字符或空格字符：isalnum、ispunct 或 space。
isupper	“A” 到 “Z” 中的一个字母。
ispunct	下述字符之一：! " # % & ' ( ) ; < = > ? [\ ] * + , - . / : ^。

定义域错误后，数学函数返回什么值？（ISO 7.5.1）

NaN。

出现下溢范围错误时，数学函数将整型表达式 errno 设置为宏 ERANGE 的值吗？（ISO 7.5.1）

是的。

当 fmod 函数有第一个为零的参数时，会出现定义域错误还是返回零？（ISO 7.5.6.4）

定义域错误。

A.16 信号

列出 signal 函数的信号集。（ISO 7.7.1.1）

表 A-6： SIGNAL 函数

名称	描述
SIGABRT	异常终止。
SIGINT	接收到交互注意信号。
SIGILL	检测到无效函数映象。
SIGFPE	错误的算术运算。
SIGSEGV	对存储的无效访问。
SIGTERM	发送到程序的终止请求。

描述一下 signal 函数识别的每个信号的使用及其参数。（ISO 7.7.1.1）

由应用定义。

描述一下 signal 函数识别的每个信号在程序启动时的默认处理和处理？（ISO 7.7.1.1）

无。

在调用信号处理程序之前，如果没有执行信号的等价 （sig,SIG_DFL），执行什么信号阻塞？ （ISO 7.7.1.1）

无。

如果为 signal 函数指定的处理程序接收到 SIGILL 信号，默认的处理被复位吗？ （ISO 7.7.1.1）

不。

A.17 流和文件

文本流的最后一行需要一个终止换行符吗？ （ISO 7.9.2）

不需要。

当读回流时，会出现写到紧接着换行符之前的文本流的空格符吗？ （ISO 7.9.2）

是的。

可向写到二进制流的数据附加多少个空字符？（ISO 7.9.2）

无。

附加模式流的文件位置指示符最初是放在文件的开始还是结束？ （ISO 7.9.3）

开始。

写文本流会导致该点之后的相关文件被截取吗？ （ISO 7.9.3）

由应用定义。

描述一下文件缓冲的特征。（ISO 7.9.3）

全缓冲。

零长度的文件实际上存在吗？ （ISO 7.9.3）

是的。

组成有效文件名的规则是什么？ （ISO 7.9.3）

由应用定义。

相同的文件可打开多次吗？ （ISO 7.9.3）

由应用定义。

remove 函数对一个打开文件有什么影响？ （ISO 7.9.4.1）

由应用定义。

如果在调用 rename 函数之前存在有新名字的函数，会有什么影响 （ISO 7.9.4.2）

由应用定义。

`fprintf` 函数中 `%p` 转换的输出形式如何？ （ISO 7.9.6.1）

十六进制标识。

`fscanf` 函数中 `%p` 转换的输入形式如何？ （ISO 7.9.6.2）

十六进制标识。

A.18 TMPFILE

如果程序异常终止，打开临时文件会被删除吗？ （ISO 7.9.4.3）

是的。

A.19 ERRNO

失败时，宏 `errno` 被 `fgetpos` 或 `ftell` 函数设置为什么值？
（ISO 7.9.9.1 和 ISO 7.9.9.4）

由应用定义。

`perror` 函数生成的消息的格式如何？ （ISO 7.9.10.4）

`perror` 的参数，后跟一个冒号，然后是值 `errno` 的文本描述。

A.20 存储器

如果请求的长度为零， `calloc`、 `malloc` 或 `realloc` 函数如何操作？
（ISO 7.10.3）

分配一个零长度的块。

A.21 ABORT

调用 `abort` 函数时，对打开和临时文件有什么影响？ （ISO 7.10.4.1）

没有影响。

A.22 EXIT

如果参数的值不是零、 EXIT_SUCCESS 或 EXIT_FAILURE 函数返回的状态如何？
（ISO 7.10.4.3）

参数的值。

A.23 GETENV

对环境名有什么限制？（ISO 7.10.4.4）

由应用定义。

描述一下用于改变调用 `getenv` 函数获得的环境列表的方法。（ISO 7.10.4.4）

由应用定义。

A.24 系统

描述传递到系统函数的字符串的格式。（ISO 7.10.4.5）

由应用定义。

系统函数执行什么执行模式？（ISO 7.10.4.5）

由应用定义。

A.25 STRERROR

描述由 `strerror` 函数输出的错误消息的格式。（ISO 7.11.6.2）

不带符号说明的字符串。

列出对 `strerror` 函数的调用返回的错误消息的内容。（ISO 7.11.6.2）

表 **A-7**： 错误消息字符串

errno	消息
`0`	没有错误
`EDOM`	定义域错误
`ERANGE`	范围错误
`EFPOS`	文件定位错误
`EFOPEN`	文件打开错误
`nnn`	错误 #nnn

注：

附录 B MPLAB C30 C 编译器诊断

B.1 简介

本附录列出了 MPLAB C30 编译器产生的最常见的诊断消息。

MPLAB C30 编译器能产生两种诊断消息：错误和警告，每种消息有不同的作用：

- 如果产生*错误*，说明程序不能通过编译。在 MPLAB C30 编译器报告的错误中，包含出错的源文件名和行号。
- 如果产生*警告*，虽然这时编译能够通过，但说明在代码中还存在一些不规范的用法，可能有问题。警告也会指出源文件名和行号，但每条消息中含有 "`warning:`" 文本，这样就可以与错误区分开来。

警告指出了程序中可能存在问题的地方，在这些地方应该确认程序是否真正表达了你的意图；或者确认是否存在无效的用法；或者确认是否存在 MPLAB C30 C 中不规范的用法。只要选择了 `-W` 选项中的一种，就会出现许多你所需要的警告（比如，`-Wall` 选项就要求多种有用的消息）。

在少数情况下，编译器可能发出内部错误消息报告。这表示编译器本身检测到一些错误，应将错误报告给 Microchip 的技术支持，具体联系方式在本手册的相应章节可以找到。

B.2 错误

符号

\x used with no following HEX digits

转义序列 \x 后应该跟十六进制数字。

'&' constraint used with no register class

asm 语句无效。

'%' constraint used with last operand

asm 语句无效。

#elif after #else

在预处理器条件编译中， #else 子句必须出现在任何 #elif 子句之后。

#elif without #if

在预处理器条件编译中，必须在使用 #elif 之前使用 #if。

#else after #else

在预处理器条件编译中， #else 子句必须仅出现一次。

#else without #if

在预处理器条件编译中，必须在使用 #else 之前使用 #if。

#endif without #if

在预处理条件编译中，必须在使用 #endif 之前使用 #if 。

#error '*message*'

出现这条错误消息是对 #error 伪指令的响应。

#if with no expression

需要一个求值为常数值的表达式。

#include expects "FILENAME" or <FILENAME>

#include 缺少文件名或文件名不完整。必须将其用引号或尖括号括起来。

'#' is not followed by a macro parameter

字符串大小操作符 "#" 必须后跟一个宏参数名。

'#*keyword*' expects "FILENAME" or <FILENAME>

指定的 #*keyword* 要求以用引号或尖括号括起来的文件名作为参数。

'#' is not followed by a macro parameter

"#" 操作符应该后跟宏参数名。

'##' cannot appear at either end of a macro expansion

连接操作符 "##" 不能出现在宏扩展的开头或结尾。

A

a parameter list with an ellipsis can't match an empty parameter name list declaration

函数的声明和定义必须一致。

"*symbol*" after #line is not a positive integer

#line 后的符号要求是一个必须为正数的源代码行号。

aggregate value used where a complex was expected

当需要复数值时，不要使用聚集值。

aggregate value used where a float was expected

当需要浮点型值时，不要使用聚集值。

aggregate valu， e used where an integer was expected

当要求整型值时不要使用聚集值。

alias arg not a string

alias 属性的参数必须是为当前标识符是别名的目标命名的字符串。

alignment may not be specified for '*identifier*'

aligned 属性只用于变量。

'__alignof' applied to a bit-field

"`__alignof`" 操作符不能用于位域。

alternate interrupt vector is not a constant

中断向量号必须为整型常量。

alternate interrupt vector number *n* is not valid

需要一个有效的中断向量号。

ambiguous abbreviation argument

指定的命令行缩写不明确。

an argument type that has a default promotion can't match an empty parameter name list declaration.

函数的声明和定义必须一致。

args to be formatted is not ...

format 属性的 first-to-check 索引参数指定了一个未声明为 “...”的参数。

argument '*identifier*' doesn't match prototype

函数参数类型应该和函数原型匹配。

argument of 'asm' is not a constant string

“asm”的参数必须为常量字符串。

argument to '-B' is missing

缺少目录名。

argument to '-l' is missing

缺少库名。

argument to '-specs' is missing

缺少 specs 文件名。

argument to '-specs=' is missing

缺少 specs 文件名。

argument to '-x' is missing

缺少语言名。

argument to '-Xlinker' is missing

缺少传递到链接器的参数。

arithmetic on pointer to an incomplete type

不允许对指向不完全类型的指针进行算术运算。

array index in non-array initializer

不要在非数组初始化中使用数组下标。

array size missing in '*identifier*'

缺少数组长度。

array subscript is not an integer

数组的下标必须为整型。

'asm' operand constraint incompatible with operand size

asm 语句无效。

'asm' operand requires impossible reload

asm 语句无效。

asm template is not a string constant

asm 模板必须为字符串常量。

assertion without predicate

#assert 或 #unassert 必须后跟一个谓词，谓词必须是一个标识符。

'*attribute*' attribute applies only to functions

属性 “*attribute*”只能用于函数。

B

bit-field '*identifier*' has invalid type

位域必须为枚举类型或整型。

bit-field '*identifier*' width not an integer constant

位域宽度必须为整型常量。

both long and short specified for '*identifier*'

变量的类型不能同时为 long 和 short。

both signed and unsigned specified for '*identifier*'

变量不能同时为 signed 和 unsigned。

braced-group within expression allowed only inside a function

函数外部的表达式内有大括号对是非法的。

break statement not within loop or switch

Break 语句只能用于循环或 switch 内部。

__builtin_longjmp second argument must be 1

`__builtin_longjmp` 要求其第二个参数为 1。

C

called object is not a function

C 中只有函数能被调用。

cannot convert to a pointer type

表达式不能转换为指针类型。

cannot put object with volatile field into register

将带有易变字段的对象存放到寄存器中是非法的。

cannot reload integer constant operand in 'asm'

asm 语句无效。

cannot specify both near and far attributes

属性 near 和 far 是互斥的，只能对函数或变量使用这两个属性中的一个。

cannot take address of bit-field '*identifier*'

试图获得位域的地址是非法的。

can't open '*file*' for writing

系统无法打开指定的 “*file*”。可能的原因包括没有足够的磁盘空间来打开文件，目录不存在，或目标目录不允许写。

can't set '*attribute*' attribute after definition

定义符号时必须使用 “*attribute*” 属性。

case label does not reduce to an integer constant

case 标号编译时必须是整型常量。

case label not within a switch statement

case 标号必须在 switch 语句内部。

cast specifies array type

指定数组类型的强制类型转换是不允许的。

cast specifies function type

指定函数类型的强制类型转换是不允许的。

cast to union type from type not present in union

不能将联合中不存在的类型强制转换为联合类型。

char-array initialized from wide string

字符数组不能用宽字符串初始化。要使用普通字符串。

***file: compiler* compiler not installed on this system**

仅安装了 C 编译器，不支持其他高级语言。

complex invalid for '*identifier*'

复数限定符只能用于整型和浮点型。

conflicting types for '*identifier*'

标识符存在多个不一致的声明。

continue statement not within loop

continue 语句只能在循环中使用。

conversion to non-scalar type requested

只能对标量类型 （不能对聚集类型）进行类型转换。

D

data type of '*name*' isn't suitable for a register

数据类型不适合请求的寄存器。

declaration for parameter '*identifier*' but no such parameter

只能声明参数列表中的参数。

declaration of '*identifier*' as array of functions

使用函数数组是非法的。

declaration of '*identifier*' as array of voids

将数组定义为 void 数组是非法的。

'*identifier*' declared as function returning a function

函数不能返回函数。

'*identifier*' declared as function returning an array

函数不能返回数组。

decrement of pointer to unknown structure

不能将指针递减到一个未知结构。

'default' label not within a switch statement

“default”case 标号必须在 switch 语句内部。

'*symbol*' defined both normally and as an alias

已定义的 “symbol”不能用作另一个符号的别名。

'defined' cannot be used as a macro name

“defined”不能用作宏名。

dereferencing pointer to incomplete type

被解引用的指针必须是指向不完全类型的指针。

division by zero in #if

被零除不可计算。

duplicate case value

case 值必须是唯一的。

duplicate label '*identifier*'

标号在其作用域内必须是唯一的。

duplicate macro parameter '*symbol*'

"symbol"在参数列表中使用多次。

duplicate member '*identifier*'

结构中不能有相同的成员。

duplicate (or overlapping) case value

case 范围不能有相同或重叠的值。错误消息 "This is the first entry overlapping that value"将提供相同或重叠的值第一次出现时的位置。 case 范围是 MPLAB C30 对 ANSI 标准的一个扩展。

elements of array '*identifier*' have incomplete type

数组元素应该有完全类型。

empty character constant

空字符常量是非法的。

empty file name in '*#keyword*'

指定为指定的 #keyword 的一个参数的文件名是空的。

empty index range in initializer

不要在初始化中使用空下标范围。

empty scalar initializer

标量初始化不能为空。

enumerator value for '*identifier*' not integer constant

枚举值必须为整型常量。

error closing '*file*'

系统不能关闭指定的 "*file*"。可能的原因包括没有足够的磁盘空间来写文件或文件太大。

error writing to '*file*'

系统不能写指定的 "*file*"。可能的原因包括没有足够的空间来写文件或文件太大。

excess elements in char array initializer

数组初始化中初值个数大于数组元素个数。

excess elements in struct initializer

结构初始化中，初值个数大于结构成员数。

expression statement has incomplete type

表达式类型不完全。

extra brace group at end of initializer

在初始化末尾，不要有多余的大括号对。

extraneous argument to '*option*' option

指定的命令行选项有过多参数。

F

‘*identifier*’ fails to be a typedef or built in type

数据类型必须为 typedef 或内建 （built-in）类型。

field ‘*identifier*’ declared as a function

指定字段不能声明为函数。

field ‘*identifier*’ has incomplete type

指定字段必须具有完全类型 （complete type）。

first argument to __builtin_choose_expr not a constant

第一个参数必须为在编译时可确定其值的常量表达式。

flexible array member in otherwise empty struct

灵活的数组元素必须是具有多个指定成员的结构的最后一个元素。

flexible array member in union

不能在联合中使用灵活的数组元素。

flexible array member not at end of struct

灵活的数组元素必须为结构的最后一个元素。

‘for’ loop initial declaration used outside C99 mode

非 C99 模式下，“for”循环初始声明无效。

format string arg follows the args to be formatted

format 属性的参数不一致。 format 字符串参数索引必须小于要检查的第一个参数的索引。

format string arg not a string type

format 属性的 format 字符串索引参数要指定一个字符串类型的参数。

format string has invalid operand number

format 属性的操作数编号参数编译时必须是一个常量。

function definition declared ‘register’

函数定义不能声明为 “register”。

function definition declared ‘typedef’

函数定义不能声明为 “typedef”。

function does not return string type

format_arg 属性只能用于返回值为字符串类型的函数。

function ‘*identifier*’ is initialized like a variable

像变量一样初始化函数是非法的。

function return type cannot be function

函数的返回值类型不能为函数。

G

global register variable follows a function definition

全局寄存器变量定义应该在函数定义之前。

global register variable has initial value

不要为全局寄存器变量赋初值。

global register variable '*identifier*' used in nested function

不要在被嵌套的函数中使用全局寄存器变量。

H

'*identifier*' has an incomplete type

指定的 "*标识符*" 有不完全类型是非法的。

'*identifier*' has both 'extern' and initializer

声明为 "extern" 的变量不能被初始化。

hexadecimal floating constants require an exponent

十六进制浮点型常量必须有指数。

I

implicit declaration of function '*identifier*'

使用函数标识符前没有原型声明或函数定义。

impossible register constraint in 'asm'

asm 语句无效。

incompatible type for argument n of '*identifier*'

在 C 中调用函数时，要确保实参类型与形参类型相匹配。

incompatible type for argument *n* of indirect function call

在 C 中调用函数时，要确保实参类型与形参类型相匹配。

incompatible types in *operation*

运算 （*操作*）中必须使用匹配的数据类型。

incomplete '*name*' option

命令行参数 *name* 的选项不完整。

inconsistent operand constraints in an 'asm'

asm 语句无效。

increment of pointer to unknown structure

不要将指针递增到一个未知结构体。

initializer element is not computable at load time

初值元素在装载时必须是可计算的。

initializer element is not constant

初值元素必须为常量。

initializer fails to determine size of '*identifier*'

数组初始化不能确定数组的大小。

initializer for static variable is not constant

要用常数初始化静态变量。

initializer for static variable uses complicated arithmetic

不要使用复杂的算术来初始化静态变量。

input operand constraint contains '*constraint*'

输入操作数的指定约束无效。

int-array initialized from non-wide string

不应该用非宽字符串初始化整型数组。

interrupt functions must not take parameters

中断函数不能接收参数。必须使用 *void* 来显式声明参数列表为空。

interrupt functions must return void

中断函数的返回值类型必须为 *void*。不允许任何其他返回值类型。

interrupt modifier '*name*' unknown

编译器要求以 "*irq*'"、"*altirq*" 或 "*save*" 作为中断属性修饰符。

interrupt modifier syntax error

中断属性修饰符有语法错误。

interrupt pragma must have file scope

#pragma 中断必须具有文件作用域。

interrupt save modifier syntax error

中断属性的 "save" 修饰符有语法错误。

interrupt vector is not a constant

中断向量号必须为整型常量。

interrupt vector number *n* is not valid

需要一个有效的中断向量号。

invalid #ident directive

#ident 应该后跟一个引号括起来的字符串常量。

invalid arg to '__builtin_frame_address'

参数应该是函数的调用函数的级别 (其中 0 产生当前函数的帧地址， 1 产生当前函数的调用函数的帧地址，等等)，为整型立即数。

invalid arg to '__builtin_return_address'

级别 (level) 参数必须为整型立即数。

invalid argument for '*name*'

编译器要求以 "data" 或 "prog" 作为 space 属性的参数。

invalid character '*character*' in #if

当一个不可打印字符 (如控制字符) 出现在 #if 后时，出现这条消息。

invalid initial value for member '*name*'

只能用整数来初始化位域 "name"。

invalid initializer

不要使用无效的初始化。

Invalid location qualifier: '*symbol*'

要求以 "*sfr*"或 "*gpr*"作为位置限定符， dsPIC DSC 器件忽略这些限定符。

invalid operands to binary '*operator*'

指定二进制操作符的操作数无效。

Invalid option '*option*'

指定的命令行选项无效。

Invalid option '*symbol*' to interrupt pragma

需要以 shadow 和 / 或 save 作为 interrupt pragma 的选项。

Invalid option to interrupt pragma

pragma 末尾有无效选项。

Invalid or missing function name from interrupt pragma

interrupt pragma 要求被调用的函数名。

Invalid or missing section name

段名必须以字母或下划线 ("_")开头，后跟一系列字母、下划线和 / 或数字。"*access*"、"*shared*"和 "*overlay*"有特殊含义。

invalid preprocessing directive #'*directive*'

不是一个有效的预处理伪指令。请检查拼写。

invalid preprologue argument

pre prologue 选项需要以汇编语句作为其参数，参数两侧要加双引号。

invalid register name for '*name*'

用无效的寄存器名将文件作用域变量 "*name*"声明为一个寄存器变量。

invalid register name '*name*' for register variable

指定的 *name* 不是一个寄存器名。

invalid save variable in interrupt pragma

要求指定要保存的有效符号。

invalid storage class for function '*identifier*'

函数不能有 "register"存储类别。

invalid suffix '*suffix*' on integer constant

整型常量只能以字母 "u"、"U"、"l"和 "L"为后缀。

invalid suffix on floating constant

浮点型常量只能以 "f"、"F"、"l"或 "L"为后缀。如果有两个 "L"，则它们必须相邻并且大小写相同。

invalid type argument of '*operator*'

operator 的参数类型无效。

invalid type modifier within pointer declarator

在指针声明中，仅可使用 `const` 或 `volatile` 作为类型修饰符。

invalid use of array with unspecified bounds

必须以有效方式使用未指定上下限的数组。

invalid use of incomplete typedef '*typedef*'

以无效的方式使用指定的 *typedef* ；这是不允许的。

invalid use of undefined type ‘*type identifier*’

以无效的方式使用*类型*；这是不允许的。

invalid use of void expression

不要使用 void 表达式。

“*name*” is not a valid filename

#line 需要有效的文件名。

‘*filename*’ is too large

指定的文件太大不能处理。文件可能大于 4 GB，处理器拒绝处理这么大的文件。要求文件大小要小于 4 GB。

ISO C forbids data definition with no type or storage class

ISO C 要求数据定义要有类型说明符或存储类别说明符。

ISO C requires a named argument before ‘...’

ISO C 要求在 “...” 前要指定参数。

L

label *label* referenced outside of any function

只能在函数内部引用标号。

label ‘*label*’ used but not defined

使用了指定的*标号*，但未定义。

language ‘*name*’ not recognized

允许的语言包括：C 和汇编。

***filename*: linker input file unused because linking not done**

命令行中指定的*文件名*，被视为链接输入文件 （因为它没有被识别为其他文件）。但链接不运行。因此，该文件被忽略。

long long long is too long for GCC

MPLAB C30 支持长度不超过 `long long` 的整型。

long or short specified with char for ‘*identifier*’

不能对字符型使用 long 和 short 限定符。

long or short specified with floating type for ‘*identifier*’

不能对浮点型使用 long 和 short 限定符。

long, short, signed or unsigned invalid for ‘*identifier*’

long、 short 和 signed 限定符只能用于整型。

M

macro names must be identifiers

宏名必须以字母或下划线开头，后跟更多字母、数字或下划线。

macro parameters must be comma-separated

参数列表中的参数之间要有逗号。

macro ‘*name*’ passed *n* arguments, but takes just *n*

传递给宏 “*name*” 的参数太多。

macro ‘*name*’ requires *n* arguments, but only *n* given

传递给宏“*name*”的参数不够。

matching constraint not valid in output operand

asm 语句无效。

‘*symbol*’ may not appear in macro parameter list

不允许将“symbol”作为参数。

Missing ‘=’ for ‘save’ in interrupt pragma

save 参数需要在列出的变量前有等号。例如：

```
#pragma interrupt isr0 save=var1,var2
```

missing ‘(’ after predicate

#assert 或 #unassert 需要在 answer 前后有括号。例如：

```
#assert PREDICATE (ANSWER)
```

missing ‘(’ in expression

括号不配对，缺少右括号。

missing ‘)’ after “defined”

缺少右括号。

missing ‘)’ in expression

括号不配对，缺少右括号。

missing ‘)’ in macro parameter list

宏要求将参数括在括号中，参数之间用逗号分隔开。

missing ‘)’ to complete answer

#assert 或 #unassert 需要在 answer 前后有括号。

missing argument to ‘*option*’ option

指定的命令行选项需要一个参数。

missing binary operator before token ‘*token*’

记号“token”前需要一个二进制操作符。

missing terminating ‘*character*’ character

缺少终止符，如单引号 '、双引号 " 或右尖括号 >。

missing terminating > character

#include 伪指令中缺少终止 >。

more than *n* operands in ‘asm’

asm 语句无效。

multiple default labels in one switch

对于每个 switch，只能指定一个 default 标号。

multiple parameters named ‘*identifier*’

参数名必须是唯一的。

multiple storage classes in declaration of ‘*identifier*’

每个声明中只能有一个存储类别。

N

negative width in bit-field ‘*identifier*’

位域宽度不能为负数。

nested function ‘name’ *declared* ‘extern’

被嵌套的函数不能声明为 “extern”。

nested redefinition of ‘*identifier*’

嵌套的重新定义是非法的。

no data type for mode ‘mode’

为 mode 属性指定的参数 mode 是可识别的 GCC 机器模式，但不是 MPLAB C30 中实现的模式。

no include path in which to find ‘*name*’

找不到包含文件 “*name*”。

no macro name given in #‘*directive*’ directive

#define、 #undef、 #ifdef 或 #ifndef 伪指令必须后跟宏名。

nonconstant array index in initializer

初始化中只能使用常量数组下标。

non-prototype definition here

如果在一个没有函数原型的函数定义后，有一个函数原型，且二者的参数个数不一致，这条消息标识出非原型定义的行号。

number of arguments doesn’t match prototype

函数的参数个数必须与函数原型相符。

O

operand constraint contains incorrectly positioned ‘+’ or ‘=’.

asm 语句无效。

operand constraints for ‘asm’ differ in number of alternatives

asm 语句无效。

operator “defined” requires an identifier

“defined”需要一个标识符。

operator ‘*symbol*’ has no right operand

预处理操作符 “symbol”的右侧需要一个操作数。

output number *n* not directly addressable

asm 语句无效。

output operand constraint lacks ‘=’

asm 语句无效。

output operand is constant in ‘asm’

asm 语句无效。

overflow in enumeration values

枚举值必须在 “int”范围内。

P

parameter ‘*identifier*’ declared void

参数不能声明为 void。

parameter ‘*identifier*’ has incomplete type

参数必须具有完全类型。

parameter ‘*identifier*’ has just a forward declaration

参数必须具有完全类型；仅有 forward 声明是不够的。

parameter ‘*identifier*’ is initialized

初始化参数是非法的。

parameter name missing

宏需要一个参数名。检查中间没有名字的两个逗号。

parameter name missing from parameter list

参数名必须包含在参数列表中。

parameter name omitted

参数名不能省略。

param types given both in param list and separately

参数类型要么在参数列表中给出，要么单独给出，但不能同时用这两种方式给出。

parse error

不能解析源代码行，它包含错误。

pointer value used where a complex value was expected

不要将指针值用于在需要复数值的地方。

pointer value used where a floating point value was expected

不要将指针值用于需要浮点值的地方。

pointers are not permitted as case values

case 值必须为整型值常量或常量表达式。

predicate must be an identifier

#assert 或 #unassert 需要一个标识符作为谓词。

predicate’s answer is empty

#assert 或 #unassert 有谓词和括号，但在括号内没有需要的答案。

previous declaration of ‘*identifier*’

这条消息标识与标识符当前声明冲突的先前声明的位置。

***identifier* previously declared here**

这条消息标识与标识符当前声明冲突的先前声明的位置。

***identifier* previously defined here**

这条消息标识与标识符当前定义冲突的先前定义的位置。

prototype declaration

标识声明函数原型的行号。与其他错误消息一起使用。

R

redeclaration of '*identifier*'

标识符 *identifier* 被声明了多次。

redeclaration of '*enum identifier*'

枚举不能被重新声明。

'*identifier*' redeclared as different kind of symbol

标识符 *identifier* 存在多次不一致的声明。

redefinition of '*identifier*'

标识符 *identifier* 被多次定义。

redefinition of '*struct identifier*'

结构不能被重新定义。

redefinition of '*union identifier*'

联合不能被重新定义。

register name given for non-register variable '*name*'

试图将一个寄存器映射到一个未标记为寄存器的变量。

register name not specified for '*name*'

文件作用域变量 "*name*" 被声明为寄存器变量，但未指定寄存器。

register specified for '*name*' isn't suitable for data type

对齐或其他限制禁止使用请求的寄存器。

request for member '*identifier*' in something not a structure or union

只有结构和联合有成员。引用其他数据类型的成员是非法的，因为其他数据类型都没有成员。

requested alignment is not a constant

aligned 属性的参数在编译时必须是常量。

requested alignment is not a power of 2

aligned 属性的参数必须是 2 的次幂。

requested alignment is too large

请求的对齐长度大于链接器的允许长度。长度必须是 4096 或更小，且是 2 的次幂。

return type is an incomplete type

返回值类型必须是完全的。

S

save variable '*name*' index not constant

数组 "name" 的下标不是整型常量。

save variable '*name*' is not word aligned

保存的对象必须是字对齐的。

save variable '*name*' size is not even

保存的对象必须是偶数长度的。

save variable '*name*' size is not known

保存的对象必须长度已知。

section attribute cannot be specified for local variables

局部变量总是分配在寄存器或堆栈中。因此试图将局部变量存放在指定的段中是非法的。

section attribute not allowed for *identifier*

段属性只能用于函数或变量。

section of *identifier* conflicts with previous declaration

如果同一个标识符 *identifier* 有多个声明指定段属性，那么属性的值必须一致。

sfr address '*address*' is not valid

地址必须小于 0x2000 才有效。

sfr address is not a constant

sfr 地址必须为常量。

'size of' applied to a bit-field

"sizeof" 不能用于位域。

size of array '*identifier*' has non-integer type

数组长度说明符必须为整型。

size of array '*identifier*' is negative

数组长度不能为负数。

size of array '*identifier*' is too large

指定的数组长度太大。

size of variable '*variable*' is too large

变量的最大长度为 32768 字节。

storage class specified for parameter '*identifier*'

不能对参数指定存储类别。

storage size of '*identifier*' isn't constant

标识符的存储长度在编译时必须为常量。

storage size of '*identifier*' isn't known

标识符 *identifier* 的长度没有完全指定。

stray '*character*' in program

源程序中不要有不需要的字符 。

strftime formats cannot format arguments

使用 format 属性时，当 archetype 参数为 strftime 时，属性的第三个参数，即指定要与格式字符串匹配的第一个参数的参数，应该为 0。 strftime 型函数没有与格式字符串匹配的输入值。

structure has no member named '*identifier*'

引用了名为 "*identifier*" 的一个结构成员，但引用的结构不包含这种成员。这是不允许的。

subscripted value is neither array nor pointer

只有数组和指针可以有下标。

switch quantity not an integer

switch 后表达式的值必须为整型。

symbol '*symbol*' not defined

在 pragma 中使用符号 "symbol" 前，要先对它进行声明。

syntax error

指定的行中存在语法错误。

syntax error ':' without preceding '?'

在 "?:" 操作符中，":" 前必须要有 "?"。

T

the only valid combination is 'long double'

double 类型前只能使用 long 限定符，不能使用其他限定符。

this built-in requires a frame pointer

`__builtin_return_address` 需要一个帧指针。不要使用 -fomit-frame-pointer 选项。

this is a previous declaration

如果标号重复，这条消息标识出前一个声明的行号。

too few arguments to function

在 C 中调用函数时，指定的参数个数不要少于函数需要的参数个数。也不能指定太多的参数。

too few arguments to function '*identifier*'

在 C 中调用函数时，指定的参数个数不要少于函数需要的参数个数。也不能指定太多的参数。

too many alternatives in 'asm'

asm 语句无效。

too many arguments to function

在 C 中调用函数时，指定的参数个数不要多于函数需要的参数个数，也不要指定太少的参数。

too many arguments to function '*identifier*'

在 C 中调用函数时，指定的参数个数不要多于函数需要的参数个数，也不要指定太少的参数。

too many decimal points in number

只需要一个小数点。

top-level declaration of '*identifier*' specifies 'auto'

只能在函数内部声明自动变量。

two or more data types in declaration of '*identifier*'

每个标识符只能有一个数据类型。

two types specified in one empty declaration

只能指定一种类型。

type of formal parameter *n* is incomplete

要为指定的参数指定一个完全的类型。

type mismatch in conditional expression

条件表达式中的类型要匹配。

typedef '*identifier*' is initialized

初始化 typedef 标识符是非法的。要使用 __typeof__。

U

‘*identifier*’ undeclared (first use in this function)

指定的标识符必须要声明。

‘*identifier*’ undeclared here (not in a function)

指定的标识符必须要声明。

union has no member named ‘*identifier*’

引用了名为 “*identifier*”的联合成员；但被引用的联合不包含这个成员。这是不允许的。

unknown field ‘*identifier*’ specified in initializer

不要在初始化中使用未知字段。

unknown machine mode ‘*mode*’

为 mode 属性指定的参数 *mode* 不是可识别的机器模式。

unknown register name ‘*name*’ in ‘asm’

asm 语句无效。

unrecognized format specifier

format 属性的参数无效。

unrecognized option ‘-*option*’

指定命令行选项无法识别。

unrecognized option ‘*option*’

“*option*”是一个未知选项。

‘*identifier*’ used prior to declaration

标识符在声明之前使用了。

unterminated #‘*name*’

要用 #endif 来终止 #if、 #ifdef 或 #ifndef 条件编译。

unterminated argument list invoking macro ‘*name*’

在完成宏扩展之前，函数宏的求值遇到了文件结束符。

unterminated comment

查找注释终止符时遇到了文件终止符。

V

‘va_start’ used in function with fixed args

“va_start”只能在有可变参数列表的函数中使用。

variable ‘*identifier*’ has initializer but incomplete type

用不完全类型初始化变量是非法的。

variable or field ‘*identifier*’ declared void

变量或字段不能被声明为 void。

variable-sized object may not be initialized

初始化长度可变的对象是非法的。

virtual memory exhausted

没有足够的存储空间来写错误消息。

void expression between '(' and ')'

需要一个常量表达式，但在括号中发现了 void 表达式。

'void' in parameter list must be the entire list

如果 "void" 作为一个参数出现在参数列表中，那么必须没有其他参数。

void value not ignored as it ought to be

不能在表达式中使用 void 函数的值。

W

warning: -pipe ignored because -save-temps specified

-pipe 选项不能和 -save-temps 选项一起使用。

warning: -pipe ignored because -time specified

-pipe 选项不能和 -time 选项一起使用。

warning: '-x *spec*' after last input file has no effect

"-x" 命令行选项仅影响命令行中此选项之后指定的文件；如果此选项前没有这种指定的文件，此选项将没有任何影响。

weak declaration of '*name*' must be public

weak 符号必须外部可见。

weak declaration of '*name*' must precede definition

"*name*" 先被定义，然后被声明为 weak。

wrong number of arguments specified for *attribute* attribute

名为 "*attribute*" 的属性有太多或太少参数。

wrong type argument to bit-complement

不要对这个运算符使用错误类型的参数。

wrong type argument to decrement

不要对这个运算符使用错误类型的参数。

wrong type argument to increment

不要对这个运算符使用错误类型的参数。

wrong type argument to unary exclamation mark

不要对这个运算符使用错误类型的参数。

wrong type argument to unary minus

不要对这个运算符使用错误类型的参数。

wrong type argument to unary plus

不要对这个运算符使用错误类型的参数。

Z

zero width for bit-field '*identifier*'

位域宽度不能为零。

B.3 警告

符号

'/*' within comment

在注释内发现了注释标记。

'$' character(s) in identifier or number

标识符名中使用美元符号是对标准的扩展。

#'*directive*' is a GCC extension

#warning、#include_next、#ident、#import、#assert 和 #unassert 伪指令是 GCC 的扩展， ISO C89 不支持这些伪指令。

#import is obsolete, use an #ifndef wrapper in the header file

#import 伪指令已废弃。#import 用于包含还没有被包含的文件。请使用 #ifndef 伪指令。

#include_next in primary source file

`#include_next` 开始在当前文件所在的目录之后搜索头文件目录列表。在这个例子中，没有先前的头文件，所以它从主源文件所在的目录开始。

#pragma pack (pop) encountered without matching #pragma pack (push, <n>)

pack （pop） pragma 必须与 pack （push） pragma 配对，在源文件中后者在前者之前。

#pragma pack (pop, *identifier*) encountered without matching #pragma pack (push, *identifier*, <n>)

pack （pop） pragma 必须与 pack （push） pragma 配对，在源文件中后者在前者之前。

#warning: *message*

伪指令 #warning 使预处理器发出一个警告并继续进行预处理。 #warning 后的记号用作警告消息。

A

absolute address specification ignored

忽略 #pragma 语句中代码段的绝对地址指定，因为 MPLAB C30 不支持它。必须在链接描述文件中指定地址，可使用关键字 `__attribute__` 来定义代码段。

address of register variable '*name*' requested

寄存器说明符禁止将地址作为变量。

alignment must be a small power of two, not n

pack pragma 的 alignment 参数必须为 2 的最小次幂。

anonymous enum declared inside parameter list

在函数参数列表内部声明了匿名枚举。良好的编程习惯是在参数列表外部声明枚举，因为当在参数列表内部定义枚举时，它们可能不会成为完全类型。

anonymous struct declared inside parameter list

在函数参数列表内部声明了匿名结构。良好的编程习惯是在参数列表外部声明结构，因为在参数列表内部定义结构时，它们可能不会成为完全类型。

anonymous union declared inside parameter list

在函数参数列表内部声明了匿名联合。良好的编程习惯是在参数列表外部声明联合，因为在参数列表内部定义联合时，它们可能不会成为完全类型。

anonymous variadic macros were introduced in C99

接受可变参数数的宏属于 C99 的一个特性。

argument '*identifier*' might be clobbered by 'longjmp' or 'vfork'

调用 longjmp 可能会改变一个参数。仅在优化编译时会出现这些警告。

array '*identifier*' assumed to have one element

指定数组的长度没有显式指定。没有这一信息时，编译器假定数组有一个元素。

array subscript has type 'char'

数组下标为 "*char*" 类型。

array type has incomplete element type

数组类型不能具有不完全元素类型。

asm operand *n* probably doesn't match constraints

指定的扩展 asm 操作数可能与其约束不匹配。

assignment of read-only member '*name*'

元素 "name" 声明为常量，但不能被赋值修改。

assignment of read-only variable '*name*'

"name" 声明为常量，但不能被赋值修改。

'*identifier*' attribute directive ignored

指定的属性不是已知或支持的属性，因此被忽略。

'*identifier*' attribute does not apply to types

指定的属性不能适用于类型，被忽略。

'*identifier*' attribute ignored

指定的属性在给定上下文中是没有意义的，因此被忽略。

'*attribute*' attribute only applies to function types

指定的属性仅适用于函数的返回值类型，不适用于其他声明。

B

backslash and newline separated by space

当处理转义序列时，遇到了反斜杠和换行符，二者之间以空格分隔。

backslash-newline at end of file

当处理转义序列时，在文件结束遇到了反斜杠和换行符。

bit-field '*identifier*' type invalid in ISO C

用于指定标识符的类型在 ISO C 中无效。

braces around scalar initializer

在初始化前后有多余的大括号对。

built-in function ‘*identifier*’ declared as non-function

指定的函数名与一内建函数名相同，但声明为其他，而没有声明为函数。

C

C++ style comments are not allowed in ISO C89

使用 C 形式的注释符 “/*” 和 “*/”，而不要使用 C++ 形式的注释符 “//”。

call-clobbered register used for global register variable

选择一个通常由函数调用保护和恢复的寄存器 （W8-W13），使库函数不会破坏它。

cannot inline function ‘main’

用 *inline* 属性声明了函数 “main”。这是不支持的，因为必须从 C 启动代码调用 main 函数，它是单独编译的。

can’t inline call to ‘*identifier*’ called from here

编译器不能内联对指定函数的调用。

case value ‘*n*’ not in enumerated type

switch 语句的控制表达式是一个枚举类型，但 case 表达式具有与任何一个枚举值都不对应的值 n。

case value ‘*value*’ not in enumerated type ‘*name*’

“*value*”是另外的 switch case，不是枚举类型 “*name*”的元素。

cast does not match function type

函数的返回值类型被强制转换为一个与函数类型不匹配的类型。

cast from pointer to integer of different size

指针被强制转换为非 16 位宽的整型。

cast increases required alignment of target type

当使用 -Wcast-align 命令行选项编译时，编译器确认强制类型转换不会增加目标类型所需要的对齐。例如，如果将指向 char 的指针强制转换为指向 int 的指针，会发出这个警告消息，因为 char 的对齐 （字节对齐）小于 int 所需的对齐 （字对齐）。

character constant too long

字符常量不能太长。

comma at end of enumerator list

枚举列表的结尾有多余的逗号。

comma operator in operand of #if

在 #if 伪指令中，不应有逗号操作符。

comparing floating point with == or != is unsafe

浮点值可近似为无限精度的实数。不检测其相等性，而使用关系操作符来看两个值的范围是否重叠。

comparison between pointer and integer

将指针类型和整型作比较。

comparison between signed and unsigned

比较的一个操作数是有符号的，另外一个操作数是无符号的。有符号操作数可视为无符号值处理，这样做可能不正确。

comparison is always *n*

比较仅包含常量表达式，因此编译器能得到比较的运行时结果。结果总是 *n*。

comparison is always *n* due to width of bit-field

由于位域的宽度，包含位域的比较的结果总是 *n*。

comparison is always false due to limited range of data type

运行时比较得结果总为 false，这是由于数据类型的范围有限。

comparison is always true due to limited range of data type

运行时比较的结果总是为 true，这是由于数据类型的范围有限。

comparison of promoted ~unsigned with constant

比较的一个操作数为提升的 ~unsigned，而另一个操作数为常数。

comparison of promoted ~unsigned with unsigned

比较的一个操作数为提升的 ~unsigned，而另一个操作数为 unsigned。

comparison of unsigned expression >= 0 is always true

比较表达式将无符号值和零做比较。由于无符号值不可能小于零，因此运行时比较的结果将总是为 true。

comparison of unsigned expression < 0 is always false

比较表达式将有符号值和零做比较。由于有符号值不可能小于零，因此运行时比较的结果将总是为 false。

comparisons like X<=Y<=Z do not have their mathematical meaning

C表达式不必和对应的数学表达式含义相同。尤其是，C表达式X<=Y<=Z并不与数学表达式 $X \leq Y \leq Z$ 等价。

conflicting types for built-in function '*identifier*'

指定的函数名字与内建函数名相同，但声明为冲突的类型。

const declaration for '*identifier*' follows non-const

指定的标识符先前声明为非 const，之后又声明为 const。

control reaches end of non-void function

非 void 函数的所有出口都应返回适当的值。编译器检测到一个非 void 函数的终止没有显式返回值的情况。因此，返回值可能是无法预估的。

conversion lacks type at end of format

当检测对 *printf*、 *scanf* 等函数的调用的参数列表时，编译器发现格式字符串中的格式字段缺少类型说明符。

concatenation of string literals with __FUNCTION__ is deprecated

对 __FUNCTION__ 的处理方式将与 __func__ （由 ISO 标准 C99 定义）相同。__func__ 是一个变量，不是字符串常量，因此它不和其他字符串常量连接。

conflicting types for '*identifier*'

指定的标识符有多个不一致的声明。

D

data definition has no type or storage class

检测到数据定义缺少类型和存储类别。

data qualifier '*qualifier*' ignored

在 MPLAB C30 中不使用数据限定符，包括 "access"、"shared" 和 "overlay"，之所以保留它们是为了与 MPLAB C17 和 C18 兼容。

declaration of '*identifier*' has 'extern' and is initialized

外部标识符不能被初始化。

declaration of '*identifier*' shadows a parameter

指定的*标识符*声明屏蔽了一个参数，使得参数不可访问。

declaration of '*identifier*' shadows a symbol from the parameter list

指定的*标识符*声明屏蔽了参数列表中的一个符号，使得该符号不可访问。

declaration of '*identifier*' shadows global declaration

指定的*标识符*声明屏蔽了一个全局声明，使得该全局符号不可访问。

'*identifier*' declared inline after being called

指定的函数在被调用之后声明为 inline。

'*identifier*' declared inline after its definition

指定的函数在被定义之后声明为 inline。

'*identifier*' declared 'static' but never defined

指定的函数声明为 static，但从未定义。

decrement of read-only member '*name*'

成员 "name" 声明为 const，不能被递减修改。

decrement of read-only variable '*name*'

"name" 声明为 const，不能被递减修改。

'*identifier*' defined but not used

指定的函数被定义了，但从未使用。

deprecated use of label at end of compound statement

标号不能位于语句的结尾。标号后应该跟随一条语句。

dereferencing 'void *' pointer

解引用 "void *" 指针是不正确的。在解引用这种指针之前，要先将指针强制转换为适当类型的指针。

division by zero

检测到编译时被零除。

duplicate ‘const’

“const”限定符只能用于声明一次。

duplicate ‘restrict’

“restrict”限定符只能用于声明一次。

duplicate ‘volatile’

“volatile”限定符只能用于声明一次。

E

embedded ‘\0’ in format

当检查对 *printf*、 *scanf* 等调用的参数列表时，编译器发现格式字符串包含 “\0”（零），这可能导致格式字符串处理过早终止。

empty body in an else-statement

else 语句为空。

empty body in an if-statement

if 语句为空。

empty declaration

声明不包含要声明的名字。

empty range specified

case 范围中的值范围为空，即低表达式的值比高表达式的值大。 case 范围的语法为 `case` *`low`* `...` *`high`*`:`。

‘enum *identifier*’ declared inside parameter list

在函数参数列表内部声明了指定的枚举。良好的编程习惯是在参数列表外部声明枚举，因为在参数列表内部定义枚举时，它们可能不会成为完全类型。

enum defined inside parms

在函数参数列表内部定义了枚举。

enumeration value ‘*identifier*’ not handled in switch

switch 语句的控制表达式为枚举类型，但并非所有的枚举值都有枚举值都有 case 表达式。

enumeration values exceed range of largest integer

枚举值用整型表示。编译器检测到一个枚举范围用任何 MPLAB C30 整型格式，包括最大的格式都无法表示。

excess elements in array initializer

初始化时提供的初值个数比声明的数组元素个数多。

excess elements in scalar initializer");

只能对标量变量初始化一次。

excess elements in struct initializer

对结构进行初始化时提供的初值数比声明的结构成员数多。

excess elements in union initializer

联合的初始化列表中的成员数比声明的联合成员数多。

extra semicolon in struct or union specified

结构类型或联合类型包含多余的分号。

extra tokens at end of #'*directive*' directive

编译器在包含 #'directive' 伪指令的源代码行中检测到多余的文本。

F

-ffunction-sections may affect debugging on some targets

如果同时指定 -g 选项和 -ffunction-sections 选项，调试时可能会遇到问题。

first argument of '*identifier*' should be 'int'

要求将指定标识符的第一个参数声明为 int 型。

floating constant exceeds range of 'double'

浮点型常量长度太大或太小，不能表示为 "double"。

floating constant exceeds range of 'float'

浮点型常量长度太大或太小，不能表示为 "float"。

floating constant exceeds range of 'long double'

浮点型常量长度太大或太小，不能表示为 "long double"。

floating point overflow in expression

当合并一个浮点型常量表达式时，编译器发现表达式溢出，即它不能表示为浮点型。

'*type1*' format, '*type2*' arg (arg '*num*')

格式的类型为 "type1"，但传递的参数类型为 "type2"。
有问题的参数为 "num" 参数。

format argument is not a pointer (arg *n*)

当检查对 *printf*、 *scanf* 等的调用的参数列表时，编译器发现指定的参数编号 *n* 不是指针，格式说明符指明它应为一个指针。

format argument is not a pointer to a pointer (arg *n*)

当检查对 *printf*、 *scanf* 等的调用的参数列表时，编译器发现指定的参数编号 *n* 不是指针，格式说明符指明它应为一个指针。

fprefetch-loop-arrays not supported for this target

目标器件不支持生成预取存储器指令的选项。

function call has aggregate value

函数的返回值为聚集。

function declaration isn't a prototype

当使用 `-Wstrict-prototypes` 命令行选项编译时，编译器确保为所有函数指定了函数原型。在本例中，遇到了之前没有函数原型的函数定义。

function declared 'noreturn' has a 'return' statement

用 noreturn 属性声明了函数没有返回值，但函数却包含返回语句。这是不一致的。

function might be possible candidate for attribute ‘noreturn’

编译器检测到函数不返回。如果用 “noreturn”属性声明了函数，那么编译器就能生成更好的代码。

function returns address of local variable

函数不应返回局部变量的地址，因为当函数返回时，局部变量占用的存储空间被释放掉了。

function returns an aggregate

函数的返回值是聚集类型。

function ‘*name*’ redeclared as inline
previous declaration of function ‘*name*’ with attribute noinline

前面用属性 noinline 声明了函数 “name”，现在又用关键字 “inline”声明了该函数来允许函数内联。

function ‘*name*’ redeclared with attribute noinline
previous declaration of function ‘*name*’ was inline

前面用 “inline”声明了函数 “name”，现在又用属性 “noinline”属性声明了该函数使函数不能被内联。

function ‘*identifier*’ was previously declared within a block

前面在一个块中显式声明了指定的函数，而在当前的代码行中又有对该函数的隐式声明。

G

GCC does not yet properly implement ‘[*]’ array declarators

编译器目前不支持长度可变的数组。

H

hex escape sequence out of range

十六进制序列必须小于十六进制数 100 （即十进制的 256）。

I

ignoring asm-specifier for non-static local variable ‘*identifier*’

当对普通的非寄存器局部变量使用 asm 说明符时，这一说明符被忽略。

ignoring invalid multibyte character

当分析一个多字节字符时，编译器确定这一字符是无效的。无效字符被忽略。

ignoring option ‘*option*’ due to invalid debug level specification

对非有效的调试级别使用了调试选项。

ignoring #pragma *identifier*

MPLAB C30 编译器不支持指定的 pragma，因此被忽略。

imaginary constants are a GCC extention

ISO C 不允许虚数数字常量。

implicit declaration of function ‘*identifier*’

指定的函数之前没有显式的声明 （函数定义或函数原型），因此编译器为其假定返回值类型和参数。

increment of read-only member ‘*name*’

元素 “name”声明为 const，不能被递增修改。

increment of read-only variable ‘*name*’

“name”声明为 const，不能被递增修改。

initialization of a flexible array member

灵活的数组元素是动态分配存储空间的，而不是静态分配的。

‘*identifier*’ initialized and declared ‘extern’

外部符号不应被初始化。

initializer element is not constant

初值元素应为常量。

inline function ‘*name*’ given attribute noinline

函数 “name”声明为 inline，但 noinline 属性使该函数不能被内联。

inlining failed in call to ‘*identifier*’ called from here

编译器不能内联对指定函数的调用。

integer constant is so large that it is unsigned

源代码中出现的整型常量值没有显式的 unsigned 修饰符，但该数字不能表示为 signed int ；因此，编译器自动将其视为 unsigned int。

integer constant is too large for ‘*type*’ type

对于 unsigned long int 型整型常量，不应超过 2^32 - 1 ；对于 long long int，不应超过 2^63 - 1 ；对于 unsigned long long int，不应超过 2^64 - 1。

integer overflow in expression

当合并一个整型常量表达式时，编译器发现表达式溢出了；即它不能表示为 int。

invalid application of ‘sizeof’ to a function type

建议不要对函数类型使用 sizeof 运算符。

invalid application of ‘sizeof’ to a void type

不应对 void 类型使用 sizeof 运算符。

invalid digit ‘digit’ in octal constant

所有数字必须在所使用进制的范围内。例如，对于八进制，只能使用数字 0 到 7。

invalid second arg to __builtin_prefetch; using zero

第二个参数必须为 0 或 1。

invalid storage class for function ‘*name*’

不应对在顶层定义的函数使用 “auto”存储类别。如果函数不是在顶层定义的，不应对函数使用 “static”存储类别。

invalid third arg to __builtin_prefetch; using zero

第三个参数必须为 0、 1、 2 或 3。

‘*identifier*’ is an unrecognized format function type

format 属性中的指定*标识符*不是可识别的格式函数类型 （`printf`、 `scanf` 或 `strftime`）之一 。

'*identifier*' is narrower than values of its type

结构的一个位域成员具有枚举类型，但位域的宽度不够，不足以表示所有枚举值。

'*storage class*' is not at beginning of declaration

指定的存储类别不在声明的开头。要求在声明中将存储类别放在最前面。

ISO C does not allow extra ';' outside of a function

在函数外部发现了多余的 "; "。这是 ISO C 不允许的。

ISO C does not support '++' and '--' on complex types

ISO C 不支持对复数类型使用递增运算符和递减运算符。

ISO C does not support '~' for complex conjugation

在 ISO C 中，不能对共扼复数使用按位取反运算符。

ISO C does not support complex integer types

ISO C 不支持复数整型，如 `__complex__` short int。

ISO C does not support plain 'complex' meaning 'double complex'

使用不加其他修饰符的 `__complex__` 与 ISO C 中不支持的 "complex double" 等价。

ISO C does not support the '*char*' '*kind of format*' format

ISO C 不支持对指定的 "格式类型" 使用说明字符 "char"。

ISO C doesn't support unnamed structs/unions

ISO C 要求 ISO C 所有的结构和 / 或联合。

ISO C forbids an empty source file

文件不包含函数或数据。这是 ISO C 不允许的。

ISO C forbids empty initializer braces

ISO C 要求将初值括在大括号内。

ISO C forbids nested functions

在一个函数内定义了另外一个函数。

ISO C forbids omitting the middle term of a ?: expression

要求在条件表达式的 "?" 和 ":" 之间要有中间项或表达式。

ISO C forbids qualified void function return type

不能对 void 函数返回值类型使用限定符。

ISO C forbids range expressions in switch statements

ISO C 不允许在单个 case 标号中指定连续值的范围。

ISO C forbids subscripting 'register' array

ISO C 不允许对 "寄存器" 数组加下标。

ISO C forbids taking the address of a label

ISO C 不允许取标号的地址。

ISO C forbids zero-size array '*name*'

"name" 数组的长度必须大于零。

ISO C restricts enumerator values to range of ‘int’

枚举值的范围不能超过 int 类型的范围。

ISO C89 forbids compound literals

符合立即数在 ISO C89 中无效。

ISO C89 forbids mixed declarations and code

应在写任何代码之前先进行声明。不要将声明和代码混合在一起。

ISO C90 does not support ‘[*]’ array declarators

ISO C90 不支持可变长度的数组。

ISO C90 does not support complex types

ISO C90 不支持复数类型，如 `__complex__ float x`。

ISO C90 does not support flexible array members

灵活的数组元素是 C99 的一个新特性。 ISO C90 不支持灵活的数组元素。

ISO C90 does not support ‘long long’

ISO C90 不支持 `long long` 类型。

ISO C90 does not support ‘static’ or type qualifiers in parameter array declarators

当使用一个数组作为函数的参数时， ISO C90 不允许数组声明使用 “static”或类型限定符。

ISO C90 does not support the ‘*char*’ ‘*function*’ format

ISO C 不支持对指定的函数格式使用说明字符 “char”。

ISO C90 does not support the ‘*modifier*’ ‘*function*’ length modifier

不支持将指定修饰符作用作给定函数的长度修饰符。

ISO C90 forbids variable-size array ‘*name*’

在 ISO C90 中，必须通过整型常量表达式来指定数组中的元素个数。

L

label ‘*identifier*’ defined but not used

定义了指定的标号，但未引用。

large integer implicitly truncated to unsigned type

源代码中的某个整型常量值没有显式的 unsigned 修饰符，但该数值不能表示为 signed int ；因此，编译器自动将这个整型常量值视为 unsigned int。

left-hand operand of comma expression has no effect

比较的一个操作数为提升的 ~unsigned，但另一个操作数为 unsigned。

left shift count >= width of type

移位位数不应小于被移位的类型中的位数。否则，移位没有意义，且结果不确定。

left shift count is negative

移位位数应该为正数。负的左移位数并不意味着右移；这是没有意义的。

library function '*identifier*' declared as non-function

指定的符号与某个库函数具有相同的名字，但被声明为其他，而没有被声明为函数。

line number out of range

C89 中 #line 伪指令的行号限制为 32767， C99 中这个值为 2147483647。

'*identifier*' locally external but globally static

指定的标识符是局部外部，但全局静态的。这样会出现问题。

location qualifier '*qualifier*' ignored

MPLAB C30 中不使用位置限定符，包括 "grp" 和 "sfr"，之所以保留是为了与 MPLAB C17 和 C18 兼容。

'long' switch expression not converted to 'int' in ISO C

ISO C 不将 "long" switch 表达式转换为 "int"。

M

'main' is usually a function

标识符 main 通常用作应用程序的主入口点的名字。编译器检测到以其他方式使用了 main，如将其作为变量的名字。

'*operation*' makes integer from pointer without a cast

指针被隐式地转换为整型。

'*operation*' makes pointer from integer without a cast

整型被隐式地转化为指针。

malformed '#pragma pack-ignored'

pack pragma 的语法不正确。

malformed '#pragma pack(pop[,id])-ignored'

pack pragma 的语法不正确。

malformed '#pragma pack(push[,id],<n>)-ignored'

pack pragma 的语法不正确。

malformed '#pragma weak-ignored'

weak pragma 的语法不正确。

'*identifier*' might be used uninitialized in this function

编译器检测到一个函数中可能使用了未初始化的指定标识符。

missing braces around initializer

初始化两边缺少大括号。

missing initializer

缺少初始化。

modification by 'asm' of read-only variable '*identifier*'

const 变量在 "asm "语句中赋值的左侧。

multi-character *character* constant

字符常量包含多个字符。

negative integer implicitly converted to unsigned type

源代码中的负整型常量值不能表示为 signed int ；因此编译器将其自动视为 unsigned int。

nested extern declaration of '*identifier*'

指定的*标识符*有嵌套的 extern 定义。

no newline at end of file

源文件的最后一行没有以换行符结束。

no previous declaration for '*identifier*'

当采用 `-Wmissing-declarations` 命令行选项编译时，编译器确保函数在定义之前声明。在本例中，遇到了之前没有函数声明的函数定义。

no previous prototype for '*identifier*'

当使用 `-Wmissing-prototypes` 命令行选项编译时，编译器确保为所有函数指定函数原型。在本例中，遇到了之前没有函数原型的函数定义。

no semicolon at end of struct or union

在结构或联合声明的结尾缺少分号。

non-ISO-standard escape sequence, '*seq*'

"seq" 为 "\e" 或 "\E"，这是对 ISO 标准的扩展。这个序列可用在字符串或字符常量中，代表 ASCII 字符 <ESC>。

non-static declaration for '*identifier*' follows static

指定的标识符先前被声明为 static，又声明为非 static。

'noreturn' function does return

用 *noreturn* 属性声明的函数返回了。这是不一致的。

'noreturn' function returns non-void value

用 *noreturn* 属性声明的函数返回了非 void 值。这是不一致的。

null format string

当检查对 *printf*、 *scanf* 等的调用的参数列表时，编译器发现缺少格式字符串。

O

octal escape sequence out of range

八进制序列必须小于八进制表示的 400 （即十进制的 256）。

output constraint '*constraint*' for operand *n* is not at the beginning

扩展 asm 中的输出约束应该在开头。

overflow in constant expression

常量表达式超出了其类型可表示的值的范围。

overflow in implicit constant conversion

隐式常量转换得到了不能表示为 signed int 的数值；因此编译器自动将其按 unsigned int 处理。

P

parameter has incomplete type

函数参数为不完全类型。

parameter names (without types) in function declaration

函数声明类出了参数的名字，但没有指定参数的类型。

parameter points to incomplete type

函数参数指向不完全类型。

parameter '*identifier*' points to incomplete type

指定的函数参数指向不完全类型。

passing arg '*number*' of '*name*' as complex rather than floating due to prototype

原型将参数 "number" 声明为复数，但使用了 float，因此编译器将其转换为复数以符合原型。

passing arg '*number*' of '*name*' as complex rather than integer due to prototype

原型将参数 "number" 声明为复数，但使用了整型值，因此编译器将其转换为复数以符合原型。

passing arg '*number*' of '*name*' as floating rather than complex due to prototype

原型将参数 "number" 声明为 float，但使用了复数值，因此编译器将其转换为 float 型以符合原型。

passing arg '*number*' of '*name*' as 'float' rather than 'double' due to prototype

原型将参数 "number" 声明为 float，但使用了 double 值，因此编译器将其转换为 float 以符合原型。

passing arg '*number*' of '*name*' as floating rather than integer due to prototype

原型将参数 "number" 声明为 float，但使用了整型值，因此编译器将其转换为 float 以符合原型。

passing arg '*number*' of '*name*' as integer rather than complex due to prototype

原型将参数 "number" 声明为整型，但使用了复数型值，因此编译器将其转换为整型以符合原型。

passing arg '*number*' of '*name*' as integer rather than floating due to prototype

原型将参数 "number" 声明为整型，但使用了 double 值，因此编译器将其转换为整型以符合原型。

pointer of type 'void *' used in arithmetic

"void" 类型的指针长度未知，不能在算术中使用。

pointer to a function used in arithmetic

不能在算术中使用指向函数的指针。

previous declaration of '*identifier*'

这条警告消息与另一条警告消息一起出现。前一条消息标识有问题代码的位置。这条消息标识*标识符*的第一次声明或定义。

previous implicit declaration of '*identifier*'

这条警告消息与警告消息 "type mismatch with previous implicit declaration" 一起出现。它指出与标识符的显式声明冲突的隐式声明的位置。

R

“*name*” re-asserted

“*name*”的答案重复。

“*name*” redefined

“name”先前已定义，现在又被再次定义。

redefinition of ‘*identifier*’

指定的标识符有多个不一致的定义。

redundant redeclaration of ‘*identifier*’ in same scope

指定的标识符在同一作用域中再次声明。这是多余的。

register used for two global register variables

两个全局寄存器变量定义为使用同一个寄存器。

repeated ‘*flag*’ flag in format

当检查对 *strftime* 的调用的参数列表时，编译器发现格式字符串中有一个重复的标志。

当检查对 *printf*、*scanf* 等的调用的参数列表时，编译器发现格式字符串中标志 { ,+,#,0,-} 之一重复。

return-type defaults to ‘int’

在没有显式的函数返回值类型声明的情况下，编译器假定函数返回 int。

return type of ‘*name*’ is not ‘int’

编译器要求 “*name*”的返回值类型为 “int”。

‘return’ with a value, in function returning void

函数声明为 void，但返回了值。

‘return’ with no value, in function returning non-void

声明为返回非 void 值的函数包含了没有值的返回语句。这是不一致的

right shift count >= width of type

移位位数应该小于被移位的类型的位数。否则移位没有意义，结果不确定。

right shift count is negative

移位位数应该为正值。负的右移位数并不意味着左移；这是没有意义的。

S

second argument of ‘*identifier*’ should be ‘char **’

要求指定标识符的第二个参数为 “char **”型。

second parameter of ‘va_start’ not last named argument

“va_start”的第二个参数必须为最后指定的参数。

shadowing built-in function ‘*identifier*’

指定的函数与某内建函数名字相同，因此屏蔽了该内建函数。

shadowing library function ‘*identifier*’

指定的函数与某个库函数名字相同，因此屏蔽了该库函数。

shift count >= width of type

移位位数应该小于被移位的类型中的位数。否则移位没有意义，结果不确定。

shift count is negative

移位位数应为正。负的左移位数并不意味着右移，负的右移位数也并不意味着左移；这是没有意义的。

size of ‘*name*’ is larger than *n* bytes

当 “name”的长度大于定义的 *len* 字节时，使用 `-Wlarger-than-len` 将发出上述警告。

size of ‘*identifier*’ is n bytes

指定标识符的长度 （*n* 字节）大于使用 `-Wlarger-than-`*`len`* 命令行选项指定的长度。

size of return value of ‘*name*’ is larger than *n* bytes

当 “name”的返回值的长度大于定义的 *len* 字节时，使用 `-Wlarger-than-len` 将发出上述警告。

size of return value of ‘*identifier*’ is n bytes

指定函数的返回值的长度为 *n* 字节，大于使用 `-Wlarger-than-`*`len`* 命令行选项指定的长度。

spurious trailing ‘%’ in format

当检查对 *printf*、 *scanf* 等的调用的参数列表时，编译器发现格式字符串中有一个伪拖尾 “%”字符。

statement with no effect

语句没有作用。

static declaration for ‘*identifier*’ follows non-static

指定的标识符在先前声明为非 static 后，又声明为 static。

string length ‘*n*’ is greater than the length ‘*n*’ ISO C*n* compilers are required to support

ISO C89 支持的最大字符串长度为 509。 ISO C99 支持的最大字符串长度为 4095。

‘struct *identifier*’ declared inside parameter list

在函数参数列表内部声明了指定的函数。良好的编程习惯是在参数列表外部声明结构，因为当在参数列表内部定义结构时，结构不能成为完全类型。

struct has no members

结构为空的，不包含成员。

structure defined inside parms

在函数参数列表内部定义了联合。

style of line directive is a GCC extension

对于传统 C，要使用格式 “#line linenum”。

subscript has type ‘char’

数组下标为类型 “*char*”。

suggest explicit braces to avoid ambiguous ‘else’

嵌套的 if 语句具有不明确的 else 子句。建议使用大括号来去除这种不明确。

suggest hiding #*directive* from traditional C with an indented #

传统 C 不支持指定的伪指令，可通过缩进 # 来 “隐藏”。
除非伪指令的 # 在第 1 列，否则伪指令被忽略。

suggest not using #elif in traditional C

不应在传统的 K&R C 中使用 #elif。

suggest parentheses around assignment used as truth value

当将赋值用作真值表时，赋值的两边要有圆括号，以使读者清楚源程序的意图。

suggest parentheses around + or - inside shift
suggest parentheses around && within ||
suggest parentheses around arithmetic in operand of |
suggest parentheses around comparison in operand of |
suggest parentheses around arithmetic in operand of ^
suggest parentheses around comparison in operand of ^
suggest parentheses around + or - in operand of &
suggest parentheses around comparison in operand of &

当在 C 中定义好了运算符优先顺序时，如果表达式的读者仅依靠优先规则而不看显式的括号的话，有时读者可能需要花一点时间来理解表达式中运算符的求值顺序。一个例子是移位中 “+” 或 “-” 运算符的使用。即使意图对于编程人员和编译器都很明确的情况下，如果使用括号来明确表明编程人员的意图的话，许多读者就不用费力来理解表达式了。

T

‘*identifier*’ takes only zero or two arguments

仅需要零或两个参数。

the meaning of ‘\a’ is different in traditional C

当使用 `-wtraditional` 选项时，转义序列 “\a” 没有被识别为 meta 序列：其值就是 “a”。在非传统编译中，“\a” 表示 ASCII BEL 字符。

the meaning of ‘\x’ is different in traditional C

当使用 `-wtraditional` 选项时，转义序列 “\x” 没有被识别为 meta 序列：其值就是 “x”。在非传统编译中，“\x” 引入一个十六进制转义序列。

third argument of ‘*identifier*’ should probably be ‘char **’

要求指定标识符的第三个参数为类型 “char **”。

this function may return with or without a value

非 void 函数的所有出口都应返回一个适当的值。编译器检测到一个非 void 函数有时有显式的返回值，有时没有。因此返回值可能是不可预估的。

this target machine does not have delayed branches

不支持 `-fdelayed-branch` 选项。

too few arguments for format

当检查对 *printf*、 *scanf* 等的调用的参数列表时，编译器发现实际参数的个数比格式字符串要求的少。

too many arguments for format

当检查对 *printf*、 *scanf* 等的调用的参数列表时，编译器发现实际参数的个数比格式字符串要求的多。

traditional C ignores #'*directive*' with the # indented

如果伪指令的 # 不在第 1 列，传统 C 忽略伪指令。

traditional C rejects initialization of unions

在传统 C 中，联合不能被初始化

traditional C rejects the 'ul' suffix

传统 C 中，后缀 “u” 无效。

traditional C rejects the unary plus operator

传统 C 中，单目加运算符无效。

trigraph ??*char* converted to *char*

三字母组合，为三字符序列，可用于表示键盘中可能没有的符号。三字母组合序列按照如下进行转换：

<table>
<tr><td>??(= [</td><td>??) =]</td><td>??< = {</td><td>??> = }</td><td>??= = #</td><td>??/ = \</td><td>??' = ^</td><td>??! = |</td><td>??- = ~</td></tr>
</table>

trigraph ??*char* ignored

三字母组合序列被忽略。 *char* 可能为 (、)、 <、 >、 =、 /、 '、 ! 或 -。

type defaults to 'int' in declaration of '*identifier*'

在指定的*标识符*没有显式的类型声明的情况下，编译器假定其类型为 int。

type mismatch with previous external decl
previous external decl of '*identifier*'

指定标识符的类型与先前的声明不匹配。

type mismatch with previous implicit declaration

显式声明与先前的隐式声明冲突。

type of '*identifier*' defaults to 'int'

当*标识符*没有显式的类型声明时，编译器假定*标识符*的类型为 int。

type qualifiers ignored on function return type

对函数返回值类型使用的类型限定符被忽略了。

U

undefining 'defined'

“defined” 不能用作宏名，不能被 undefine。

undefining '*name*'

对先前定义的宏名 “name” 使用了 #undef 伪指令。

union cannot be made transparent

对联合使用了 `transparent_union` 属性，但指定的变量不满足该属性的要求。

'union *identifier*' declared inside parameter list

在函数参数列表内部声明了指定的联合。良好的编程习惯是在参数列表外部声明联合，因为当在参数列表内部定义联合时，联合不能成为完全类型。

union defined inside parms

在函数参数列表内部定义了联合。

union has no members

联合是空的，不包含成员。

unknown conversion type character '*character*' in format

当检查对 *printf*、 *scanf* 等的调用的参数列表时，编译器发现格式字符串中的一个转换字符是无效的 （不可识别）。

unknown conversion type character 0x*number* in format

当检查对 *printf*、 *scanf* 等的调用的参数列表时，编译器发现格式字符串中的一个转换字符是无效的 （不可识别）。

unknown escape sequence '*sequence*'

“sequence”不是一个有效的转义代码。转义代码必须以 “\”开头，并使用下列字符之一：n、 t、 b、 r、 f、 b、 \、 '、 "、 a 或 ?，或者它必须为八进制或十六进制数字。如果为八进制数字，数字序列必须小于八进制的 400。如果是十六进制数字，数字序列必须以 “x”开头，必须小于十六进制的 100。

unnamed struct/union that defines no instances

结构 / 联合是空的，且没有名字。

unreachable code at beginning of *identifier*

在指定函数的开头有不可到达的代码。

unrecognized gcc debugging option: *char*

“char”对于 -d*letters* 调试选项不是有效的字母。

unused parameter '*identifier*'

指定的函数参数在该函数中未使用。

unused variable '*name*'

声明了指定的变量，但未使用。

use of '*' and 'flag' together in format

当检查对 *printf*、 *scanf* 等的调用的参数列表时，编译器发现格式字符串中出现了标志 “*”和 “flag”。

use of C99 long long integer constants

ISO C89 中不允许将整型常量声明为 long long。

use of '*length*' length modifier with '*type*' type character

当检查对 *printf*、 *scanf* 等的调用的参数列表时，编译器发现对指定的*类型*错误地使用了指定的长度。

'*name*' used but never defined

使用了指定的函数，但从未定义。

'*name*' used with '*spec*' '*function*' format

“name”对于指定函数的格式中的转换说明无效。

useless keyword or type name in empty declaration

空声明包含无用的关键字或类型名。

V

__VA_ARGS__ can only appear in the expansion of a C99 variadic macro

应将预定义的宏 `__VA_ARGS` 用于使用省略号的宏定义的替换部分中。

value computed is not used

一个计算的值未使用。

variable '*name*' declared 'inline'

只能对函数使用关键字 "inline"。

variable '%s' might be clobbered by 'longjmp' or 'vfork'

非 volatile 的自动变量可能被 longjmp 调用更改。仅在优化编译时才可能出现这些警告 。

volatile register variables don't work as you might wish

将一个变量作为参数传递可能将变量传递到另外的寄存器 (W0-W7)中，而不是为参数传递指定的寄存器 (如果不是 W0-W7)中。或者编译器可能发出一条不适合于指定寄存器的指令，并可能需要暂时将值移动到其他地方仅当指定的寄存器被异步修改 (即通过中断服务程序)时才会有问题。

W

-Wformat-extra-args ignored without -Wformat

为使用 `-Wformat-extra-args`，必须指定 `-Wformat`。

-Wformat-nonliteral ignored without -Wformat

为使用 `-Wformat-nonliteral`，必须指定 `-Wformat`。

-Wformat-security ignored without -Wformat

为使用 `-Wformat-security`，，必须指定 `-Wformat`。

-Wformat-y2k ignored without -Wformat

为使用 -Wformat-y2k，必须指定 `-Wformat`。

-Wid-clash-LEN is no longer supported

不再支持 `-Wid-clash-LEN`。

-Wmissing-format-attribute ignored without -Wformat

为使用 `-Wmissing-format-attribute`，必须指定 `-Wformat`。

-Wuninitialized is not supported without -O

为使用 `-Wuninitialized`，必须启用优化。

'*identifier*' was declared 'extern' and later 'static'

指定的标识符先前被声明为 "extern"，现在又被声明为 static。

'*identifier*' was declared implicitly 'extern' and later 'static'

指定的标识符先前被隐式声明为 "extern"，现在又被声明为 static。

'*identifier*' was previously implicitly declared to return 'int'

存在一个与先前的隐式声明的不匹配。

'*identifier*' was used with no declaration before its definition

当使用 -Wmissing-declarations 命令行选项编译时，编译器确保在定义函数前声明函数。在本例中，遇到了事先没有函数声明的函数定义。

'*identifier*' was used with no prototype before its definition

当使用 -Wmissing-prototypes 命令行选项编译时，编译器确保为所有函数指定函数原型。在这种情况下，遇到了被调用函数事先没有函数原型的函数调用。

writing into constant object (arg *n*)

当检查对 *printf*、 *scanf* 等的调用的参数列表时，编译器发现指定参数编号 *n* 是一个常量对象，格式说明符指明要写这个常量对象。

Z

zero-length *identifier* format string

当检查对 *printf*、 *scanf* 等的调用的参数列表时，编译器发现格式字符串是空的 （“”）。



附录 C MPLAB C18 与 MPLAB C30 C 编译器

C.1 简介

本章的目的是着重介绍 MPLAB C18 和 MPLAB C30 C 编译器的区别。关于 MPLAB C18 编译器的更多细节，请参阅 *《MPLAB® C18 C 编译器用户指南》*（DS51288J_CN）。

本章讨论两种编译器在以下方面的区别：

- 数据格式
- 指针
- 存储类别
- 堆栈使用
- 存储限定符
- 预定义宏名
- 整型的提升
- 字符串常量
- 匿名结构
- 快速存取存储区
- 行内汇编
- Pragma 伪指令
- 存储模型
- 调用约定
- 启动代码
- 编译器管理的资源
- 优化
- 目标模块格式
- 实现定义的操作
- 位域

C.2 数据格式

表 C-1： 数据格式中的位数

数据格式	MPLAB® C18(1)	MPLAB® C30(2)
`char`	8	8
`int`	16	16
`short long`	24	-
`long`	32	32
`long long`	-	64
`float`	32	32
`double`	32	32 或 64(3)

注 1： MPLAB C18 使用自己的数据格式，类似于 IEEE-754 格式，但最高九位有所不同（参见表 C-2）。
2： MPLAB C30 使用 IEEE-754 格式。
3： 参见第 **5.5** 节 “浮点型”。

表 C-2： **MPLAB® C18** 浮点型与 **MPLAB C30 IEEE-754** 格式

标准	字节 **3**	字节 **2**	字节 **1**	字节 **0**
MPLAB C30	seeeeeee1	$e_0$ddd dddd16	dddd dddd8	dddd dddd0
MPLAB C18	eeeeeeee0	sddd dddd16	dddd dddd8	dddd dddd0

其中：s = 符号位， d = 尾数， e = 指数

C.3 指针

表 C-3： 指针的位数

存储器类型	MPLAB® C18	MPLAB® C30
程序存储器－ Near	16	16
程序存储器－ Far	24	16
数据存储器	16	16

C.4 存储类别

MPLAB C18 允许对变量使用非 ANSI 的存储类别修饰符 `overlay`，对函数参数使用非 ANSI 的存储类别修饰符 `auto` 或 `static`。

MPLAB C30 不允许这些修饰符。

C.5 堆栈使用

表 C-4： 使用的堆栈类型

堆栈中的项	MPLAB® C18	MPLAB® C30
返回地址	硬件	软件
局部变量	软件	软件

C.6 存储限定符

MPLAB C18 使用非 ANSI 的 far、 near、 rom 和 ram 类型限定符。

MPLAB C30 使用非 ANSI 的 far、 near 和 space 属性。

例 **C-1**： 定义 **NEAR** 变量

```
C18: near int gVariable;
C30: __attribute__((near)) int gVariable;
```

例 **C-2**： 定义 **FAR** 变量

```
C18: far int gVariable;
C30: __attribute__((far)) int gVariable;
```

例 **C-3**： 生成在程序存储器中的变量

```
C18: rom int gArray[6] = {0,1,2,3,4,5};
C30: __attribute__((section(".romdata"), space(prog) ))
       int gArray[6] = {0,1,2,3,4,5};
```

C.7 预定义宏名

MPLAB C18 定义了 __18CXX， __18F242，……（所有其他以 __ 为前缀的处理器），以及 __SMALL__ 或 __LARGE__，这取决于所选择的存储模型。

MPLAB C30 定义了 __dsPIC30。

C.8 整型的提升

MPLAB C18 按照最大操作数的大小执行整型的提升，即使两个操作数长度都小于 int。 MPLAB C18 提供了 -Oi+ 选项来与标准符合。

MPLAB C30 按照 ISO 的要求，以 int 精度或更高精度执行整型的提升。

C.9 字符串常量

MPLAB C18 将字符串常量保存在程序存储器中的 .stringtable 段中。MPLAB C18 支持字符串函数的几种形式。例如， strcpy 函数有四种形式，允许将字符串从数据存储器和程序存储器拷贝到数据存储器和程序存储器。

MPLAB C30 通过 PSV 窗口来访问数据存储器或程序存储器中的字符串常量，允许像访问任何其他数据一样访问常量。

C.10 匿名结构

MPLAB C18 支持非 ANSI 的联合内匿名结构。

MPLAB C30 不支持匿名结构。

C.11 快速存取存储区

dsPIC30F 器件没有快速存取存储区。

C.12 行内汇编

MPLAB C18 使用非 ANSI 的 `_asm` 和 `_endasm` 来标识行内汇编程序块。

MPLAB C30 使用非 ANSI 的 `asm`，它看起来很像函数调用。MPLAB C30 中 `asm` 语句的使用在第 **8.4** 节 **"使用行内汇编"** 中详述。

C.13 PRAGMA 伪指令

MPLAB C18 使用 pragma 伪指令来声明段 （`code`、 `romdata`、 `udata` 和 `idata`）、中断 （高优先级和低优先级）和变量分配 （存储区、段）。

MPLAB C30 使用非 ANSI 的属性，不使用 pragma 伪指令。

表 C-5： MPLAB® C18 PRAGMA 伪指令和 MPLAB C30 属性

Pragma （MPLAB® C18）	属性 （MPLAB® C30）
`#pragma udata [name]`	`__attribute__ ((section ("name")))`
`#pragma idata [name]`	`__attribute__ ((section ("name")))`
`#pragma romdata [name]`	`__attribute__ ((space (prog)))`
`#pragma code [name]`	`__attribute__ ((section ("name"))),` `__attribute__ ((space (prog)))`
`#pragma interruptlow`	`__attribute__ ((interrupt))`
`#pragma interrupt`	`__attribute__ ((interrupt, shadow))`
`#pragma varlocate bank`	NA*
`#pragma varlocate name`	NA*

*dsPIC® DSC 器件存储器不分区。

例 C-4： 指定未初始化变量位于数据存储器中的用户段

```
C18: #pragma udata mybss
     int gi;
C30: int __attribute__((__section__(".mybss"))) gi;
```

例 C-5： 将变量 MABONGA 分配到数据存储器中的地址 0X100

```
C18: #pragma idata myDataSection=0x100;
     int Mabonga = 1;
C30: int __attribute__((address(0x100))) Mabonga = 1;
```

例 **C-6**： 指定将一个变量存放到程序存储器中

C18:
```
#pragma romdata const_table
const rom char my_const_array[10] =
  {0,1,2,3,4,5,6,7,8,9};
```

C30:
```
const __attribute__((space(const)))
char my_const_array[10] = {0,1,2,3,4,5,6,7,8,9};
```

注：	MPLAB C30 不直接支持对程序空间中变量的访问。这样分配的变量必须由编程人员显式访问，通常通过使用表访问行内汇编指令，或使用程序空间可见性窗口来访问。关于 PSV 窗口的更多信息，请参阅**第 4.15 节 "程序空间可视性 （PSV）的使用"**。

例 **C-7**： 将函数 **PRINTSTRING** 分配到程序存储器中的地址 **0X8000**

C18:
```
#pragma code myTextSection=0x8000;
int PrintString(const char *s){...};
```

C30:
```
int __attribute__((address(0x8000))) PrintString
  (const char *s) {...};
```

例 **C-8**： 编译器自动保护和恢复变量 **VAR1** 和 **VAR2**

C18:
```
#pragma interrupt isr0 save=var1, var2
void isr0(void)
{
/* perform interrupt function here */
}
```

C30:
```
void
__attribute__((__interrupt__(__save__(var1,var2))))
isr0(void)
{
/* perform interrupt function here */
}
```

C.14 存储模型

MPLAB C18 使用非 ANSI 的小存储模型和大存储模型。小存储模型使用 16 位指针并将程序存储器限制为小于 64 KB （32 K 字）。

MPLAB C30 使用非 ANSI 的小代码模型和大代码模型。小代码模型将程序存储器限制为小于 96 KB （32 K 字）。在大代码模型中，指针可能使用跳转表。

C.15 调用约定

MPLAB C18 和 MPLAB C30 的调用约定有许多差别。关于 MPLAB C30 调用约定的论述，请参阅**第 4.12 节 "函数调用约定"**。

C.16 启动代码

MPLAB C18 提供了三个启动子程序 — 一个启动子程序不对用户数据进行初始化，一个启动子程序仅初始化有显式初始化的变量，另外一个启动子程序初始化所有变量（按照 ANSI 标准的要求，将无显式初始化的变量初始化为零）。

MPLAB C30 提供两个启动子程序 — 一个启动子程序不对用户数据进行初始化，另外一个启动子程序初始化除持久数据段中的变量之外的所有变量（按照 ANSI 标准的要求，将无显式初始化的变量初始化为零）。

C.17 编译器管理的资源

MPLAB C18 有如下编译器管理的资源：PC、 WREG、STATUS、 PROD、段 .tmpdata、段 MATH_DATA、 FSR0、 FSR1、 FSR2、 TBLPTR 和 TABLAT。

MPLAB C30 有如下编译器管理的资源：W0-W15、 RCOUNT 和 SR。

C.18 优化

下面列出了每个编译器进行的优化。

MPLAB® C18	MPLAB® C30
转移优化 （-Ob+） 代码排序 （-Os+） 尾部合并 （-Ot+） 删除执行不到的代码 （-Ou+） 复制传递 （-Op+） 冗余存储删除 （-Or+） 死代码删除 （-Od+）	优化设置 （-On，其中 n 为 1，2，3 或 s）[1]
合并相同的字符串 （-Om+）	-fwritable-strings
存储区选择优化 （-On+）	无 — 不使用存储区
W 寄存器内容跟踪 （-Ow+）	自动跟踪所有寄存器
过程抽象 （-Opa+）	过程抽象 （-mpa）

注 **1：** 在 MPLAB C30 中，这些优化设置可满足大部分需要。可使用其他选项来进行 "微调"。更多信息参见**第 3.5.6 节 "控制优化的选项"**。

C.19 目标模块格式

MPLAB C18 和 MPLAB C30 使用不同的 COFF 文件格式，文件格式不可互换。

C.20 实现定义的操作

对于负的整型值的右移：

- MPLAB C18 不保留符号位
- MPLAB C30 保留符号位

C.21 位域

MPLAB C18 中的位域不能跨越字节存储边界，因此长度不能大于 8 位。

MPLAB C30 支持任意位数的位域，位域长度最大为基础类型的长度。任何整型都可以作为一个位域。分配不能跨越基础类型的位边界。

例如：

```
struct foo {
  long long i:40;
  int j:16;
  char k:8;
} x;

struct bar {
  long long I:40;
  char J:8;
  int K:16;
} y;
```

使用 MPLAB C30，结构 foo 的大小将为 10 个字节。 i 将被分配到位偏移 0 处 （直到位偏移 39）。在 j 之前会填充 8 位，分配在位偏移 48 处。如果将 j 分配到下一个可用的位偏移 40 处，那么对于一个 16 位的整型，将跨越一个存储边界。将在 j 之后分配 k，分配到位偏移 64 处。结构的最后包含 8 位的填充，是为了保持为数组时需要的对齐。对齐为 2 字节，因为结构中的最大对齐为 2 字节。

使用 MPLAB C30，结构 bar 的大小将为 8 个字节。 I 将被分配到位偏移 0 处 （直到位偏移 39）。 J 之前不需要填充，因为对于一个 char，不会跨越存储边界。 J 被分配到位偏移 40。 K 可从位偏移 48 开始分配，这样就完成了整个结构的分配，没有浪费任何存储空间。

注：

附录 D 不赞成使用的特性

D.1 简介

这里论述的特性视为过时，且已被更高级的功能代替。包含不赞成使用特性的项目在所述的语言工具版本中将正常工作。使用不赞成使用的特性将导致警告；建议编程人员修改项目来去除不赞成使用的特性。在语言工具的将来版本中将完全取消对这些特性的支持。

D.2 主要内容

不赞成使用的特性包括：

- 预定义常量

D.3 预定义常量

MPLAB® C30 C 编译器定义了如下预处理符号。

符号	由 **-ansi** 命令行选项定义？
`dsPIC30`	否
`__dsPIC30`	是
`__dsPIC30__`	是

编译器的 ELF 版本定义了如下预处理符号。

符号	由 **-ansi** 命令行选项定义？
`dsPIC30ELF`	否
`__dsPIC30ELF`	是
`__dsPIC30ELF__`	是

编译器的 COFF 版本定义了如下预处理符号。

符号	由 **-ansi** 命令行选项定义？
`dsPIC30COFF`	否
`__dsPIC30COFF`	是
`__dsPIC30COFF__`	是

最新信息请参阅**第 3.7 节 “预定义常量”**。

注：



附录 E ASCII 字符集

表 E-1： **ASCII 字符集**

最高有效字符

最低有效字符

Hex	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	NUL	DLE	Space	0	@	P	‘	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	Bell	ETB	’	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

注：



附录 F GNU 自由文档许可证

GNU 自由文档许可证 （GNU Free Documentation License）
版本 1.2， 2002 年 11 月

0. 前言

本许可证用于使手册、教材或其他功能性的有用的文档获得 “自由 （Free）”：保证任何人确实可以自由地拷贝与分发经过或未经改动的该文档，无论是否用于商业目的。其次，本许可证使作者和发行人获得其作品的信誉，而不需要对其他人的改动负责。

本许可证是一种 “Copyleft”，这意味着文档的衍生作品必须也是同样自由的。本许可证是为自由软件设计的Copyleft 许可证GNU通用公共许可证（General Public License，GPL）的补充。

我们为自由软件的手册的使用设计了本许可证，因为自由软件需要自由的文档：一个自由的程序应随一个提供同样自由的文档一同发布。但本许可证不仅仅限于软件的手册，还可用于任何文本作品，无论它的主题是什么或它是否作为印刷的书籍而发行。我们建议将这个许可证主要用于说明或参考性的文献作品。

1. 适用性和定义

本许可证适用于任何包含版权所有者的一份声明，宣布该作品在本许可证的条款下可以发布的使用任何介质的手册或其他作品。这个声明授予在此处声明的条件下，无限期、无版税地在全球使用作品的许可。下述的 “文档” 指任何这样的手册或作品。任何公众中的一员都是被授予许可的人，这里称为 “您”。如果您在版权法规定的需要许可的情况下复制、修改或分发作品，则表明您接受了本许可证。

文档的 “修改版本 （Modified Version）” 意味着包含文档或文档的一部分的作品，它可以是逐字复制，也可以是进行了修改和 / 或翻译成为了其他语言。

“次要章节（Secondary Section）” 是文档中特定的附录或引言章节，专用于处理文档的发行人或作者与文档的总体主题（或相关问题）的关系，不包含任何在总体主题中介绍的内容。（这样，如果文档是一本数学教材，“次要章节” 可能不包含任何数学内容。）这种关系可能是与主题或相关问题的历史关联，或者是关于主题或相关问题的法律、商业、哲学、伦理或政治立场。

“不可变章节 （Invariant Section）” 是某些次要章节，其标题在文档以本许可证发布的声明中被指定为不可变章节。一个章节如果不符合上述次要章节的定义，就不能指定为不可变章节。文档可以没有不可变章节。如果文档没有指定不可变章节，则视为没有不可变章节。

“封皮文本 （Cover Text）” 是一些在文档按照本许可证发布的声明中被列为封面文本（Front-Cover Text）或封底文本 （Back-Cover Text）的一些短段落。封面文本最多 5 个词，封底文本最多 25 个词。

文档的 “透明 （Transparent）” 拷贝是指机器可读的拷贝，使用公众可以得到其规范的格式表达，适用于使用通用文本编辑器、通用绘图程序（对于由像素构成的图像）或被广泛使用的绘画程序（对于绘制的图形）直接修改文档；也适用于输入到文本格式处理程序或自动转换成各种适于输入到文本格式处理程序的格式。用其他透明文件格式表示的拷贝，如果该格式的标记（或缺少标记）已经构成了对读者后续修改的障碍，那么就不是透明的。如果用一个图像格式表示确实有效的文本，不论数量多少，都不是透明格式。不 “透明”的拷贝称为 “不透明 （Opaque）”。

透明拷贝的适用格式的例子有：没有标记的纯 ASCII 文本、 Texinfo 输入格式、 LaTeX 输入格式、使用公众可用的 DTD 的 SGML 或 XML，符合标准的简单 HTML 以及可以手工修改的 PostScript 或 PDF。透明的图像格式的例子有 PNG、XCF 和 JPG。不透明格式包括：仅可被私有版权的字处理软件读取和编辑的私有版权格式、所用的 DTD 和 / 或处理工具不是广泛可用的 SGML 或 XML，机器生成的 HTML，某些字处理器生成的只用于输出目的的 PostScript 或 PDF。

对于印刷书籍，“扉页”（Title Page）指扉页本身，以及随后的一些需要用来明显地保存本许可证所要求出现在扉页上的材料的页。对于那些没有这类扉页的作品形式，“扉页”指接近作品最突出的标题的、在文本正文之前的文本。

“标题为 XYZ （Entitled XYZ）”的章节指文档的特定子单元，其标题就是 XYZ，或是将 XYZ 翻译为其他语言的文本之后的括号中包含 XYZ。（这里， XYZ 指下面提及的特定章节的名字，如 “致谢 （Acknowledgement）”、“献给 （Dedication）”、“签名（Endorsement）”或 “历史 （History）”。）在修改文档时对这些章节 “保护标题（Preserve the Title）”意思是依据这个定义保留这样一个 “标题为 XYZ”的章节。

文档可在本许可证适用于该文档的声明后面包含免责声明（Warranty Disclaimer）。这些免责声明被认为是以引用的方式包括在本许可证中，但这仅被视为放弃担保：免责声明中任何其他暗示都是无效的，并对本许可证的含义没有影响。

2. 逐字复制

您可以以任何介质拷贝并分发文档，无论是否出于商业目的，只要保证本许可证、版权声明和宣称本许可证适用于文档的声明都在所有拷贝中被完整而无任何附加条件地给出即可。您不能使用任何技术手段阻碍或控制对您制作或发布的拷贝的阅读或再次复制。但您可以在复制品的交易中取得报酬。如果您发布足够多的拷贝，必须遵循第 3 节中的条件。

您也可以在与上述相同的条件下出租拷贝和向公众展示拷贝。

3. 大量复制

如果您发行文档的印刷版拷贝 （或是有印制封皮的其他介质的拷贝）多于 100 份，而文档的许可证声明中要求有封皮文本，就必须将拷贝用封皮封起来，并在封皮文本中清晰明显地记载以下文字：封面文本在封面上，封底文本在封底上。封面和封底上还必须清晰明显地标明您是这些拷贝的发行者。封面必须同等显著、可见地完整展现标题的所有文字。您可以在标题上加入其他材料。改动仅限于封皮的复制，只要保持文档的标题不变并满足这些条件，可以在其他方面被视为逐字复制。

如果所需文本对于封面或封底过多，无法明显地表示，您应该在封皮上列出前面的几条（在合理的前提下尽量多），将其他的放在邻近的页面上。

如果您出版或分发了超过 100 份文档的不透明拷贝，您必须在每个不透明拷贝中包含一份机器可读的透明拷贝，或是在每个不透明拷贝中或另行给出一个计算机网址，通过该网址，使用计算机网络的公众可以使用标准的网络协议没有任何附加条件地下载一个文档的完整的透明拷贝。如果您选择后者，就必须在开始大量分发不透明拷贝时采取相当谨慎的步骤，以保证透明拷贝在所给出的网址从您最后一次 （直接或通过代理和零售商）向公众分发该版本的不透明拷贝之后至少一年始终是有效的。

在重新大量发布拷贝之前，请您（但不是必须）与文档的作者联系，以使他们有机会向您提供文档的更新版本。

4. 修改

在上述第 2、 3 节的条件下，您可以复制与分发文档的修改版本，只要严格按照本许可证发布修改版本，并用修改版本取代原有文档，这样就许可了任何得到修改版本的拷贝的人分发或修改这个修改后的版本。另外，在修改版本中，您必须做到如下几点：

a) 在扉页 （及其他封皮，如果有的话）中使用与文档及先前各个版本 （如果有的话，应该被列在文档的历史章节中）不同的标题。如果那个版本的原始发行者允许，您可以使用与该版本相同的标题。
b) 在扉页上列出作为作者的承担修改版本中修改工作的一个或多个人或实体和至少五个文档的原作者 （如果原作者不足五个就全部列出），除非他们免除了您的这一责任。
c) 在扉页上列出修改版本的发行者的名字作为发行者。
d) 保持文档的全部版权声明不变。
e) 在与其他版权声明邻近的位置加入恰当的针对您所作修改的版权声明。

f) 包含一份未作任何修改的本协议的拷贝。
g) 保持命名为 “历史 （History）” 的章节不变，保持其标题不变，并在其中加入一项，至少声明扉页上的修改版本的标题、年、新作者和新发行者。如果文档中没有名为 “历史” 的章节，就新建这一章节，并加入声明原文档扉页上所列的标题、年、作者与发行者的项，并在后面加入如上句所述的描述修改版本的项。
h) 如果文档中有供公众访问的文档透明拷贝的网址，保持网址不变，并同样把它所基于的以前版本的网址给出。这些网址可以放在 “历史” 章节中。您可以省略在本文档发行之前至少四年已经发表的作品的网址，或者在网址所指的版本的原始发行者允许时，也可不列出网址。
i) 对于任何以名为 “致谢” 或 “献给” 的章节，保持章节的标题不变，并保持章节中对每位供稿人的感谢和 / 或献给的全部内容和语气不变。
j) 保持文档的所有不可变章节不变，不改变其内容和标题。章节的编号或同等信息不视为章节标题的一部分。
k) 删除任何名为 “签名 （Endorsement）” 的章节。修改版本中可以不包含这类章节。
l) 不要将任一已经存在的章节重命名为 “签名” 或与任何不可变章节的名字相冲突的名字。
m) 保留所有免责声明不变。

如果修改版本加入了新的符合次要章节定义的引言或附录章节，并且不包含任何从原文档中复制的内容，您就可以按照您的意愿将它标记为不可变。如果需要这样做，就把它们的标题加入修改版本的许可声明中的不可变章节列表之中。这些标题必须区别于其他章节的标题。

您可以加入一个名为 “签名” 的章节，只要它只包含各方对您的修改版本给出的签名——例如书评或是声明文本已经被某个组织认定为某一标准的权威定义。

您可以在修改版本的封皮文本列表的末尾加入一个最多五个词的段落作为封面文本，和一个最多 25 个词的段落作为封底文本。任一实体只可加入 （或通过排列制作） 一段封面或封底文本。如果原文档已经为该封皮（封面或封底）包含了封皮文本，先前由您或您所代表的实体加入或制作的文本，就不能再新加入一个，但您可以在原发行者的明确许可下替换掉原来的那个。

文档作者和发行者不因本许可证而授权将他们的名字用于对任何修改版本的宣传、表示或暗示认可。

5. 合并文档

遵照以上第 4 节所述的修改版本的规定，您可以将文档和其他按照本许可证发布的文档合并，只要您在合并结果中包含原始文档的所有不可变章节，对它们不加以任何改动，并在合并作品的许可声明中将它们全部列为不可变章节，而且保持原作者的所有免责声明不变。

合并的作品仅需要包含一份本许可证，多个相同的不可变章节可以由一个来取代。如果有多个名称相同而内容不同的不可变章节，通过在章节名称后面添加用括号括起的原作者、发行者的名字（如果知道的话）来加以区别，或通过惟一的编号加以区别。在合并作品的许可声明中须将不可变章节列表中的章节标题做相同的修改。

在合并过程中，您必须合并不同原始文档中任何名为“历史”的章节，从而形成一个新的名为 “历史”的章节；类似地，还要合并名为 “致谢”和 “献给”的章节。您必须删除所有名为 “签名”的章节。

6. 文档的合集

您可以制作一个文档和其他按照本许可证发布的文档的合集，并在合集中将不同文档中的多个本许可证的拷贝替换为一个单独的拷贝，只要您在文档的其他方面遵循本许可证的逐字复制的规定。

您可以从这样的合集中提取一个单独的文档，并将它按照本许可证下单独发布，只要您向这个提取出来的文档中加入一份本许可证的拷贝，并在文档的其他方面遵循本许可证的逐字复制的规定。

7. 独立作品的聚合体

文档或文档的衍生品和其他与之相分离的独立文档或作品汇编成一个存储体或放在分发介质上，如果汇编后的版权对汇编作品的使用者的权利的限制没有超出原来的独立作品的许可范围，称为文档的 “聚合体 （Aggregate）”。当以本许可证发布的文档被包含在一个聚合体中时，本许可证不适用于聚合体中非该文档的衍生作品的其他作品。

如果第 3 节中的封皮文本的要求适用于文档的拷贝，那么如果文档在聚合体中所占的比重小于全文的一半，文档的封皮文本可以被放置在聚合体内包含该文档部分的封皮上，或者如果文档是电子形式的，那么封皮文本就被放置在电子文档的等效部分。否则，它必须位于整个聚合体的印刷的封皮上。

8. 翻译

翻译被认为是一种修改，因此您可以按照第 4 节中的规定发布文档的翻译版本。如果要将文档的不可变章节用翻译版取代，需要得到著作权人的允许，但您可以将部分或全部不可变章节的翻译版附加在原始版本的后面。您可以包含一个本许可证、本文档的所有许可证声明和免责声明的翻译版本，只要您同时包含他们的原始英文版本、许可证声明和免责声明的原始版本。当本许可证、许可证声明和免责声明的翻译版本和英文版不一致时，以原始版本为准。

在文档的名为 “致谢”、“献给” 和 “历史” 的章节中，第 4 节中的保持标题不变的要求将是要更换实际的标题。

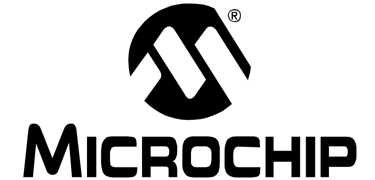
9. 终止

除非遵照本许可证明确说明，否则您不可以对按照本许可证发布的文档进行复制、修改、附加许可证或发布。任何其他试图复制、修改、附加许可和发布本文档的行为都是无效的，并自动终止本许可证所授予您的权利。然而，从您这里按照本许可证得到的拷贝或权力的人（或组织），只要他们仍然完全遵守本许可证，得到的许可证将不会终止。

10. 本许可证的未来修订版本

Free Software Foundation （FSF）可能会不时发布 GNU 自由文档许可证的新的修订版本。这些新版本的主旨将与现在的版本类似，但可能在解决某些问题和利害关系的细节上有所不同。请参见 http://www.gnu.org/copyleft/ 。

本许可证的每个版本都有惟一的版本号。如果文档指定本许可证的某一特定版本“或任何之后的版本”为适用的许可证，您可以选择遵循指定版本或 FSF 发布的任何之后的版本 （非草案）的条款和条件。如果文档没有指定本许可证的版本，那么您可以选择遵循任何 FSF 曾经发布的版本 （非草案）。



术语表

ANSI

美国国家标准委员会是美国负责指定和批准标准的组织。

ASCII

美国信息交换标准码是使用 7 个二进制数字来表示每个字符的字符集编码。它包括大写和小写字母、数字、符号以及控制字符。

标识符 （Identifier）

函数名或变量名。

表达式 （Expression）

由算术操作符或逻辑操作符分隔开的常量和 / 或符号的组合。

C

具有表达式简练、现代控制流程和数据结构，以及运算符丰富特点的通用编程语言。

COFF

公共目标文件格式。这种格式的目标文件包含机器代码、调试和其他信息。

操作码 （Opcode）

操作码。*参见*助记符。

程序存储器 （Program Memory）

器件中存储指令的存储器。

程序计数器 （Program Counter）

包含正在执行的指令的地址的存储单元。

重定位 （Relocation）

链接器执行的一个过程，将绝对地址分配到可重定位段，且将可重定位段中的所有符号更新到新地址。

存储类别 （Storage Class）

确定对象的生存时间。

存储模型 （Memory Model）

应用程序可使用的存储器的表示。

存储限定符 （Storage Qualifier）

表明对象的特殊属性 （如 `volatile`）。

错误 （Error）

错误报告使程序不能继续执行的问题。如果可能，错误标识出存在问题的源文件名和行号。

DSC

*参见*数字信号控制器。

DSP

*参见*数字信号处理器。

单片机 （Microcontroller， MCU）

包含 CPU、 RAM、程序存储器、 I/O 口和定时器的高度集成芯片。

递归调用 （Recursive Call）

直接或间接调用本身的函数。

地址 （Address）

标识存储器中一个单元的值。

段 （Section）

指定的代码或数据序列。

堆 （Heap）

用于动态存储空间分配的存储区，在此存储区中，在运行时以任意顺序分配和释放存储块。

堆栈，软件 （Stack, Software）

由应用程序使用，用于存储返回地址、函数参数和局部变量的存储空间。当用高级语言开发代码时这个存储空间通常由编译器管理。

Endianess

描述多字节对象中字节的存放顺序。

Epilogue

编译器生成代码的一部分，负责释放堆栈空间、恢复寄存器及满足运行时模型中指定的任何其他特定于机器的要求。这段代码在给定函数的任何用户代码之后、紧接着函数返回之前执行。

Free-standing

一种编译器实现，它接受任何不使用复杂类型的严格符合程序，而且在这种实现中，对 ISO 库条款中指定的特性的使用，仅限于标准头文件 `<float.h>`、`<iso646.h>`、 `<limits.h>`、 `<stddef.h>` 和 `<stdint.h>` 的内容。

GPR

通用寄存器。器件数据存储器的一部分。

高级语言 （High Level Language）

用于编写比汇编更脱离处理器机器代码的程序的语言。

归档 （Archive）

可重定位目标模块的集合。由将多个源文件汇编为目标文件，然后使用归档器将目标文件组合为一个库文件生成。可将库与目标模块和其他库链接，生成可执行代码。

归档器 （Archiver）

生成和操作库的工具。

国际标准化组织 （International Organization for Standardization）

指定许多行业和技术 （包括计算和通讯）方面的标准的一个组织。

Hex 代码 （Hex Code）

存储在十六进制格式代码中的可执行指令。 hex 代码包含在 hex 文件中。

Hex 文件 （Hex File）

包含可对器件编程的十六进制地址和值 （hex 代码）的 ASCII 文件。

宏 （Macro）

宏指令。以缩写形式表示一系列指令的指令。

汇编器 （Assembler）

将汇编语言源代码翻译为机器代码的语言工具。

汇编语言 （Assembly Language）

以符号形式描述二进制机器代码的编程语言。

IDE

集成开发环境。 MPLAB IDE 是 Microchip 的集成开发环境。

IEEE

电子和电气工程师协会

IRQ

*参见*中断请求。

ISO

*参见*国际标准化组织。

ISR

*参见*中断服务程序。

机器代码 （Machine Code）

处理器实际读和解释的计算机程序的表示。二进制机器代码的程序由一系列机器指令（可能还包含数据）组成。特定处理器的所有可能指令的集合称为 “指令集”。

机器语言 （Machine Language）

特定中央处理单元的指令集，不需翻译即可用于处理器。

警告 （Warning）

警告报告表明存在问题但并未停止处理的条件。在 MPLAB C30 中，警告消息报告源文件名和行号，但包含文本 “`warning`:” 以与错误消息区分开。

可重定位 （Relocatable）

其段未被分配到固定的存储地址的目标文件。

可执行代码 （Executable Code）

已准备好装入执行的软件。

库 （Library）

*参见*归档。

库管理器 （Librarian）

*参见*归档器。

快速访问存储区 （Access Memory）（仅 PIC18）

PIC18XXXXX 器件的特殊寄存器，对这些寄存器的访问与存储区选择寄存器 （Bank Select Register， BSR）的设置无关。

Little Endianess

多字节数据的一种数据存放顺序方案，其中最低字节存放在较低地址。

链接描述文件 （Linker Script File）

链接器的命令文件。定义链接选项并描述目标平台上的可用存储器。

链接器 （Linker）

组合目标文件和库生成可执行代码，并解析从一个模块到另一个模块的引用的语言工具。

MPLAB ASM30

Microchip dsPIC30F 数字信号控制器的可重定位宏汇编器。

MPLAB C1X

指 Microchip 的 MPLAB C17 和 MPLAB C18 C 编译器。MPLAB C17 是 PIC17CXXX 器件的 C 编译器， MPLAB C18 是 PIC18CXXX 和 PIC18FXXXX 器件的 C 编译器。

MPLAB C30

Microchip dsPIC30F 数字信号控制器的 C 编译器。

MPLAB IDE

Microchip 的集成开发环境。

MPLAB LIB30

MPLAB LIB30 归档器 / 库管理器是目标库管理器，用于使用 MPLAB ASM30 或 MPLAB C30 C 编译器生成的 COFF 目标模块。

MPLAB LINK30

MPLAB LINK30 是 Microchip MPLAB ASM30 汇编器和 Microchip MPLAB C30 C 编译器的目标链接器。

命令行接口 （Command Line Interface）

仅基于文本输入和输出，在程序和其用户之间进行通讯的一种方式。

目标文件 （Object File）

包含机器代码并可能包含调试信息的文件。可能是可直接执行的文件，或是需要与其他目标文件 （如库）链接生成完整的可执行程序的可重定位文件。

匿名结构 （Anonymous Structure）

未命名结构。

PICmicro MCU

PICmicro 单片机 （MCU）指 Microchip 的所有单片机系列。

Pragma

对特定编译器有意义的伪指令。 pragma 伪指令通常用来将实现定义的信息传达给编译器。 MPLAB C30 使用属性来传达这一信息。

Prologue

编译器生成代码的一部分，负责释放堆栈空间、保存寄存器及满足运行时模型中指定的任何其他特定于机器的要求。这段代码在给定函数的任何用户代码之前执行。

器件编程器 （Device Programmer）

用于对电可编程半导体器件 （如单片机）进行编程的工具。

RAM

随机访问存储器 （数据存储器）。这种存储器中的信息可以按任意顺序访问。

ROM

只读存储器 （程序存储器）。不能修改的存储器。

软件模拟器 （Simulator）

模拟器件操作的软件程序。

SFR

*参见*特殊功能寄存器。

三字母组合 （Trigraph）

三个字符组成的序列，均以 ?? 开头，由 ISO C 定义为单个字符的替代。

属性 （Attribute）

C 程序中用于描述特定于机器的属性的变量或函数的特性。

数据存储器 （Data Memory）

在 Microchip MCU 和 DSC 器件中，数据存储器 （RAM）由通用寄存器 （GPR）和特殊功能寄存器 （SFR）组成。某些器件还有 EEPROM 数据存储器。

数字信号控制器 （Digital Signal Controller）

具有数字信号处理能力的单片机 （即 Microchip 的 dsPIC DSC 器件）。

数字信号处理 （Digital Signal Processing）

数字信号及已转换为数字形式 （被采样）的普通模拟信号 （声音或图像）的计算机处理。

数字信号处理器 （Digital Signal Processor）

设计为用于数字信号处理的微处理器。

特殊功能寄存器 （Special Function Registers）

数据存储器（RAM）中专门用于控制 I/O 处理器功能、 I/O 状态、定时器或其他模式或外设的部分。

伪指令 （Directive）

源代码中控制语言工具操作的语句。

未初始化数据 （Uninitialized Data）

定义时未提供初始值的数据。在 C 中，

```
int myVar;
```

定义了将存放在未初始化数据段的一个变量。

文件寄存器 （File Register）

片内数据存储器，包括通用寄存器 （GPR）和特殊功能寄存器 （SFR）。

向量 （Vector）

当发生特定事件 （如复位或中断）时，应用程序开始执行处的存储地址。

已初始化数据 （Initialized Data）

用初始值定义的数据。在 C 中，

```
int myVar=5;
```

定义了一个位于已初始化数据段中的变量。

运行时模型 （Runtime Model）

描述目标架构资源的使用。

优先顺序 （Precedence）

定义表达式中求值顺序的规则。

源代码 （Source Code）

编程人员编写计算机程序的形式。采用某种正式的编程语言编写源代码，可翻译为机器代码或被解释程序执行。

源文件 （Source File）

包含源代码的 ASCII 文本文件。

运算符 （Operator）

构成表达式时使用的符号，如加法符号 “+”和减法符号 “-”。每个运算符都有用于确定求值顺序的指定优先顺序。

帧指针 （Frame Pointer）

指向堆栈中地址的指针，它将基于堆栈的参数和基于堆栈的局部变量分隔开。提供了一个方便的途径来访问当前函数的局部变量和其他值。

指令 （Instruction）

告知中央处理单元 （CPU）执行特定操作，并包含操作中要使用的数据的一系列位。

指令集 （Instruction Set）

特定处理器理解的机器语言指令的集合。

中断 （Interrupt）

发送到 CPU 的信号，暂停正在运行的应用程序，并将控制权转交给中断服务程序来处理事件。

中断处理程序 （Interrupt Handler）

发生中断时处理特殊代码的子程序。

中断服务程序 （Interrupt Service Routine）

发生中断时调用的函数。

中断请求 （Interrupt Request）

处理器暂停正常的指令执行并开始执行中断处理程序的事件。某些处理器有几种中断请求事件，允许具有不同的中断优先级。

中断响应延迟时间 （Latency）

中断事件与其响应之间的时间。

助记符 （Mnemonics）

可直接翻译为机器代码的文本指令。也称为操作码。

字母数字 （Alphanumeric）

字母数字字符由字母字符和十进制数字 （0,1, …, 9）组成。

字母字符 （Alphabetic Character）

字母字符指阿拉伯字母 （a, b, …, z, A, B, …, Z）字符。

左值 （L-value）

引用可被检查和 / 或修改的对象的表达式。左值表达式用在赋值的左侧。

注：





索引

K

L

M

N

O

P

Q

R

S

X

Y

注：



全球销售及服务网点

美洲

公司总部 Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 1-480-792-7200
Fax: 1-480-792-7277
技术支持：
http://support.microchip.com
网址：www.microchip.com

亚特兰大 Atlanta
Alpharetta, GA
Tel: 1-770-640-0034
Fax: 1-770-640-0307

波士顿 Boston
Westborough, MA
Tel: 1-774-760-0087
Fax: 1-774-760-0088

芝加哥 Chicago
Itasca, IL
Tel: 1-630-285-0071
Fax: 1-630-285-0075

达拉斯 Dallas
Addison, TX
Tel: 1-972-818-7423
Fax: 1-972-818-2924

底特律 Detroit
Farmington Hills, MI
Tel: 1-248-538-2250
Fax: 1-248-538-2260

科科莫 Kokomo
Kokomo, IN
Tel: 1-765-864-8360
Fax: 1-765-864-8387

洛杉矶 Los Angeles
Mission Viejo, CA
Tel: 1-949-462-9523
Fax: 1-949-462-9608

圣何塞 San Jose
Mountain View, CA
Tel: 1-650-215-1444
Fax: 1-650-961-0286

加拿大多伦多 Toronto
Mississauga, Ontario,
Canada
Tel: 1-905-673-0699
Fax: 1-905-673-6509

亚太地区

中国 – 北京
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

中国 – 成都
Tel: 86-28-8676-6200
Fax: 86-28-8676-6599

中国 – 福州
Tel: 86-591-8750-3506
Fax: 86-591-8750-3521

中国 – 香港特别行政区
Tel: 852-2401-1200
Fax: 852-2401-3431

中国 – 青岛
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

中国 – 上海
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

中国 – 沈阳
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

中国 – 深圳
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

中国 – 顺德
Tel: 86-757-2839-5507
Fax: 86-757-2839-5571

中国 – 武汉
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

中国 – 西安
Tel: 86-29-8833-7250
Fax: 86-29-8833-7256

台湾地区 – 高雄
Tel: 886-7-536-4818
Fax: 886-7-536-4803

台湾地区 – 台北
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

台湾地区 – 新竹
Tel: 886-3-572-9526
Fax: 886-3-572-6459

亚太地区

澳大利亚 Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

印度 India - Bangalore
Tel: 91-80-4182-8400
Fax: 91-80-4182-8422

印度 India - New Delhi
Tel: 91-11-5160-8631
Fax: 91-11-5160-8632

印度 India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

日本 Japan - Yokohama
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

韩国 Korea - Gumi
Tel: 82-54-473-4301
Fax: 82-54-473-4302

韩国 Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 或
82-2-558-5934

马来西亚 Malaysia - Penang
Tel: 60-4-646-8870
Fax: 60-4-646-5086

菲律宾 Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

新加坡 Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

泰国 Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

欧洲

奥地利 Austria - Wels
Tel: 43-7242-2244-399
Fax: 43-7242-2244-393

丹麦 Denmark-Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

法国 France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

德国 Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

意大利 Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

荷兰 Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

西班牙 Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

英国 UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

02/16/06